(2)

March 1990 UILU-ENG-90-2209

COORDINATED SCIENCE LABORATORY
College of Engineering

SEQUENTIAL DECODING WITH ADAPTIVE REORDERING OF CODEWORD TREES

Branko Radosavljevic

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2209	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-90-J-1270
8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Sequential Decoding with Adaptive Reordering of Codeword Trees

12. PERSONAL AUTHOR(S)
Radosavljevic, Branko

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ________ TO ______	March 1990	137

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	low density codes, sequential decoding, computational cut-off rate

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

are presented for

We present modifications ~~of~~ standard sequential decoding algorithms in an attempt to operate at rates greater than R_0, the computational cutoff rate. We call the new algorithms sequential decoding with reordering (SDR) algorithms. They observe thc received message at the channel output and use this information to reorder the digits in the codeword tree. The resulting tree is then searched by a sequential decoder; the goal of reordering is to obtain a tree that is easy to search. However, codeword trees associated with convolutional codes cannot be reordered and still retain their uniform structure and slow growth. For this reason, we use low-density codes, a class of block codes.

The SDR algorithms are presented for the binary erasure and binary symmetric channels. Simulation results suggest that at high code rates, the algorithms can be used at rates nearly equal to or greater than R_0. [illegible]

(continued)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

19. Abstract (continued)

Because of differences between codeword trees for low-density codes and convolutional codes, ~~we present~~ a new sequential decoding algorithm designed for low-density codes. ~~We also present~~ an SDR algorithm that can be used as a soft decision decoder on channels with side information and channels with real-valued outputs. [illegible]

SEQUENTIAL DECODING WITH ADAPTIVE REORDERING OF CODEWORD TREES

BY

BRANKO RADOSAVLJEVIC

B.S., University of Illinois, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

MARCH 1990

WE HEREBY RECOMMEND THAT THE THESIS BY

BRANKO RADOSAVLJEVIC

ENTITLED SEQUENTIAL DECODING WITH ADAPTIVE REORDERING OF CODEWORD TREES

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

[illegible] Director of Thesis Research

Timothy N. Trick Head of Department

Committee on Final Examination†

Chairperson

† Required for doctor's degree but not for master's.

O-517

ABSTRACT

We present modifications of standard sequential decoding algorithms in an attempt to operate at rates greater than R_0, the computational cutoff rate. We call the new algorithms sequential decoding with reordering (SDR) algorithms. They observe the received message at the channel output and use this information to reorder the digits in the codeword tree. The resulting tree is then searched by a sequential decoder; the goal of reordering is to obtain a tree that is easy to search. However, codeword trees associated with convolutional codes cannot be reordered and still retain their uniform structure and slow growth. For this reason, we use low-density codes, a class of block codes.

The SDR algorithms are presented for the binary erasure and binary symmetric channels. Simulation results suggest that at high code rates, the algorithms can be used at rates nearly equal to or greater than R_0.

Because of differences between codeword trees for low-density codes and convolutional codes, we present a new sequential decoding algorithm designed for low-density codes. We also present an SDR algorithm that can be used as a soft decision decoder on channels with side information and channels with real-valued outputs.

DEDICATION

To my father, my mother, and my brother Alexander.

ACKNOWLEDGEMENTS

I would like to thank my two advisors, Professor Erdal Arikan and Professor Bruce Hajek, for their invaluable assistance and guidance. In particular, Professor Arikan provided conceptual insight and the original idea behind SDR algorithms, and Professor Hajek provided insight as well as many important improvements.

I am also grateful for the support of the Office of Naval Research, through their graduate fellowship program.

I would like to thank my fellow students in the Communication Group, a truly excellent group of people.

Finally, I would like to thank my family, for their patience, help, and support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

We present modifications of standard sequential decoding algorithms in an attempt to operate at rates greater than the computational cutoff rate. Sequential decoding is a general method for decoding tree codes, including the important class of convolutional codes. It was invented by Wozencraft in 1957 and subsequently modified by Fano [4] and others. For a good introduction to this topic, see Section 7.2 of Clark and Cain [3].

The amount of computation performed by a sequential decoder depends on the channel noise. In particular, a long burst of noise will require a great deal of computation. This is illustrated in the following example, taken from [8]. Suppose we have a binary erasure channel with transition probabilities as shown in Figure 1.1. Suppose further that we use a convolutional code with rate R. If the first L received symbols are erasures, each path with length L symbols in the codeword tree will appear equally likely to the decoder until the paths are extended further into the tree. There are approximately 2^{LR} such paths, and each path has probability 1/2 of being extended before the decoder finds the correct one. Thus, given L initial erasures, the expected number of paths searched by the decoder is $2^{LR}/2$.

A long burst of noise will cause a sequential decoder to perform a great deal of computation even on more general communication channels. For

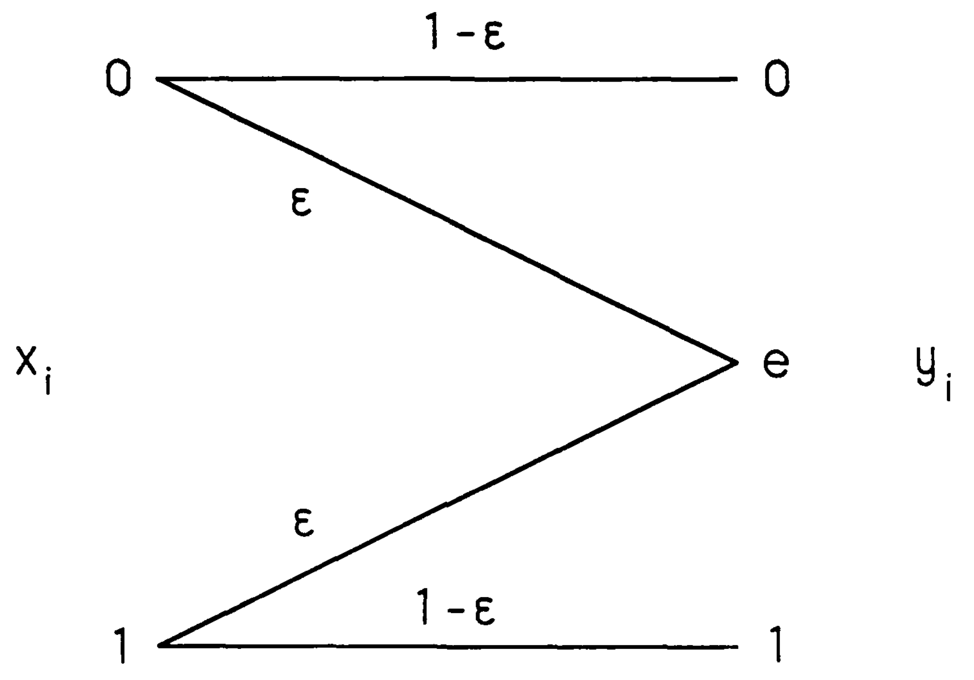


Figure 1.1. The binary erasure channel.

example, consideration of bursts allowed Jacobs and Berlekamp [8] to prove the following result. Consider a discrete memoryless channel with input alphabet $\{0,1,\ldots,K-1\}$, output alphabet $\{0,1,\ldots,J-1\}$, and transition probabilities $P(j\,|\,k)$. If one uses a code with rate R satisfying $R > R_\rho$, then

$$\lim_{n\to\infty} \mathrm{E}(c_n^\rho) = \infty\,, \tag{1.1}$$

where

$$R_\rho = \frac{\hat{E}_0(\rho)}{\rho}$$

$$\hat{E}_0(\rho) = \text{the concave hull of } E_0(\rho)$$

$$E_0(\rho) = \text{Gallager's reliability exponent}$$
$$= \max_Q \left\{ -\log \sum_{j=0}^{J-1} \left[\sum_{k=0}^{K-1} Q(k) P(j\,|\,k)^{1/(1+\rho)} \right]^{1+\rho} \right\}$$

$$c_n = \text{the amount of computation per symbol required to decode the first } n \text{ symbols}$$

This result is valid for all $\rho > 0$, for all tree codes, which include the class of convolutional codes, and for all sequential decoders. It is derived by considering the effect of noise bursts with specified length and severity. These noise bursts are shown to occur sufficiently often and cause the decoder to search enough of the codeword tree to result in the unbounded moments of computation described above. Note that Arikan [1] has obtained an improved bound for the case $\rho=1$. His result shows that $\mathrm{E}(c_n) = \infty$ for all $R > E_0(1)$, which differs from the result given above by using $E_0(\rho)$ instead of its concave hull.

For future reference, note that $E_0(1)$, which is a function of the channel, is traditionally denoted by R_0. (This presentation follows [1].) The computational cutoff rate for sequential decoding, R_{comp}, is defined to be the supremum of code rates for which $\overline{\lim_{n\to\infty}} \mathrm{E}(c_n) < \infty$. From Arikan's result and previous work (p. 279 of Gallager [5]), it is known that $R_{comp} = R_0$. We use R_0 as a reference point to gauge the performance of the decoding algorithms presented in this work.

The preceding discussion leads one to consider what would happen if bursts of noise could somehow be eliminated. Would it be possible to have bounded moments of computation even at rates above R_ρ? One way to limit the occurrence and duration of bursts is to adaptively reorder the codeword tree, that is, to change the order of the digits used to generate the tree. However, one cannot significantly reorder the codeword tree associated with a convolutional code and still obtain a tree that is practical to search.

Instead of using convolutional codes, one can use low-density codes. Low-density codes, devised by Gallager [6], [7], are linear block codes characterized by three parameters. A binary (n, j, k) low-density code has blocklength n, and a parity matrix with exactly j 1's in each column and k 1's in each row. All low-density codes considered in this paper are binary.

Typically, j and k are much smaller than n, resulting in a sparse parity matrix. In addition, choosing small values for j and k causes low-density codes to have the following two features. First, given any ordering of the parity checks used to define a low-density code, one can generate a codeword tree that grows slowly compared to the growth for a dense parity matrix. Second, different parity check orderings yield different codeword trees.

These two features are used by the decoding algorithms presented in this thesis. The algorithms, which we call sequential decoding with reordering (SDR) algorithms, all have the structure shown in Figure 1.2. The ordering algorithm observes the channel output and uses this information to choose a parity check ordering. The sequential decoder then searches the corresponding codeword tree to obtain the decoder output.

A method for generating codeword trees for low-density codes with given parity check orderings is presented in Section 1.2. Some differences between these trees and codeword trees for convolutional codes are discussed. Section 1.3 summarizes some of the properties of low-density codes. This includes a result concerning their asymptotic minimum distance, obtained by Gallager [7]. Section 1.4 describes some known decoding algorithms for low-density codes.

Chapter 2 discusses an SDR algorithm for binary erasure channels. It includes the results of using this algorithm on a simulated communication channel.

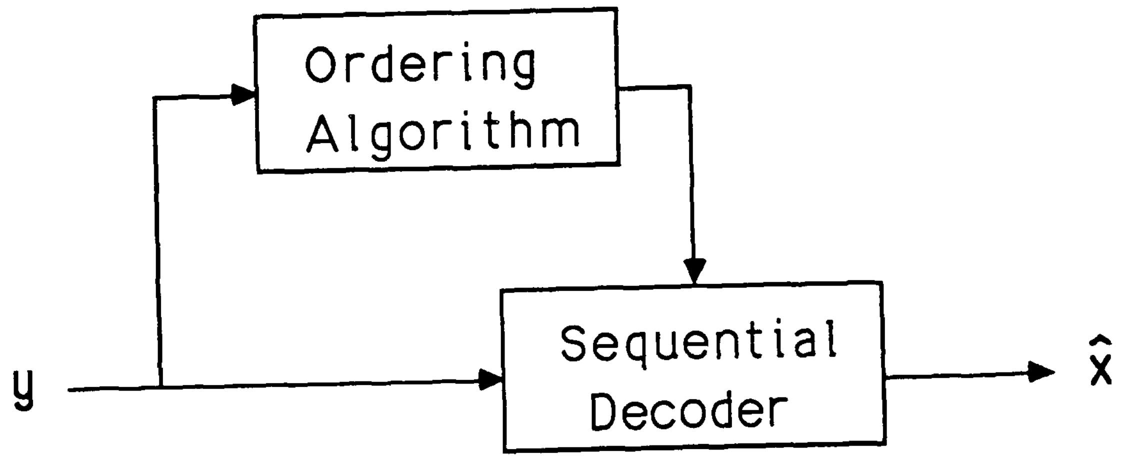


Figure 1.2. The general structure of SDR algorithms.

Chapter 3 contains SDR algorithms for binary symmetric channels. Several ordering algorithms are presented. Because of the differences between codeword trees for low-density codes and convolutional codes, a new sequential decoding algorithm is presented as well. Simulation results are presented also.

Chapter 4 contains conclusions and possible directions for further research. A decoding algorithm for memoryless channels with binary input and arbitrary output is presented. In particular, it is applicable to additive white Gaussian noise channels and Rayleigh fading channels. This algorithm allows one to use soft decision decoding.

Throughout this work we use the following terminology. The vector $\mathbf{x} = (x_i)_{i=1}^n$ is the transmitted codeword, $\mathbf{y} = (y_i)_{i=1}^n$ is the channel output, or received message, and $\hat{\mathbf{x}} = (\hat{x}_i)_{i=1}^n$ is the decoder output, where n is the block-length of the code being used.

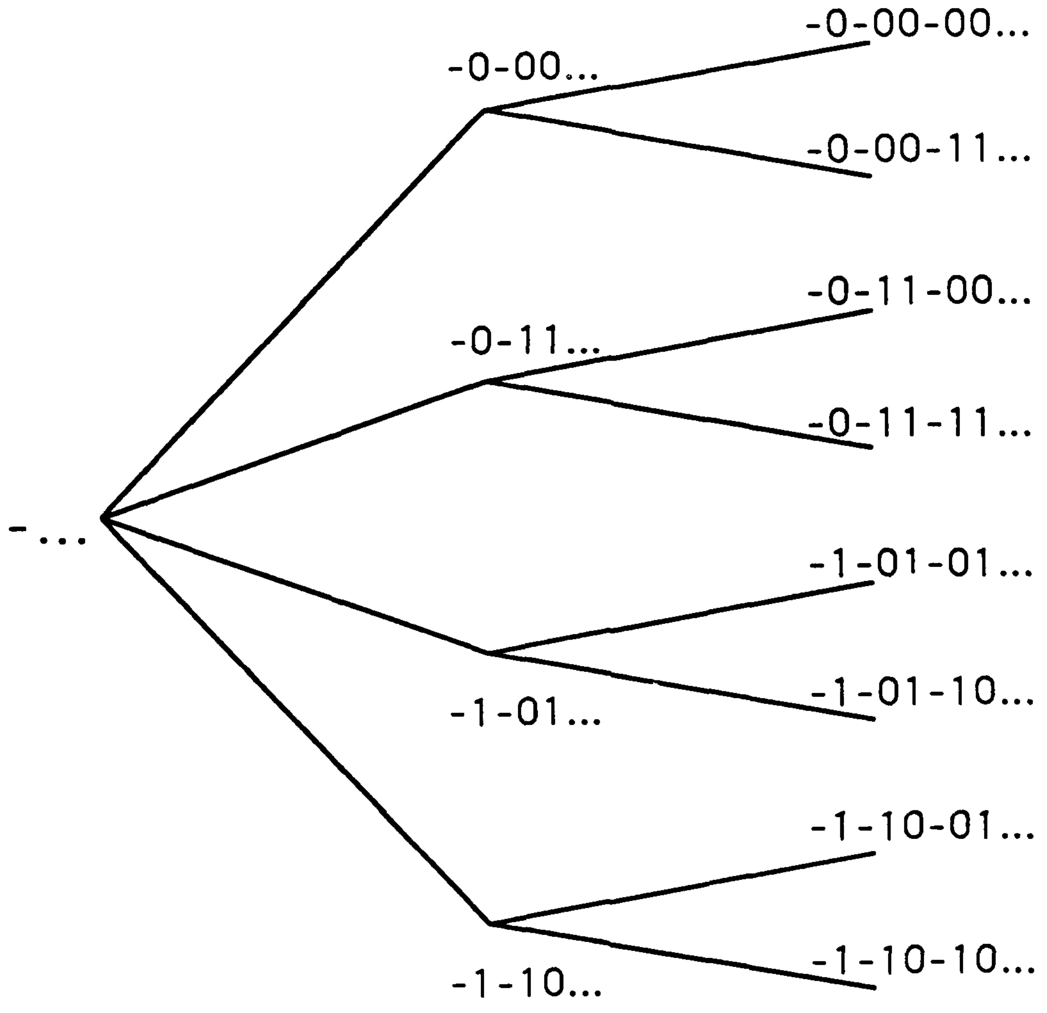
1.2 Codeword Trees for Low-Density Codes

In this section, we present a method for generating codeword trees for low-density codes and discuss some properties of these trees.

The following description applies to a low-density code with parameters n, j, and k. To generate a codeword tree, one must first assign an ordering to the parity checks used to define the code. Any ordering will yield a code-word tree, and at this point we assume the ordering is arbitrary. Algorithms for choosing an ordering are given in Chapters 2 and 3. Each parity check corresponds to a level in the codeword tree. Since there are nj/k parity checks, there will be nj/k levels in the tree. As with any tree code, the nodes of the tree correspond to partially filled codewords. The root node is completely unfilled and the nodes at the final level correspond to complete codewords.

The first level of the tree is generated by assigning values to the digits involved in the first parity check. A different node is created for each assignment that satisfies the parity check. (See Figure 1.3.) Since there are k digits involved in each parity check, there will be 2^{k-1} nodes in the first level. Each of these nodes is extended to the second level by assigning values to the digits involved in the second parity check. Again, this is done in all ways that satisfy the new parity check. However, if some digits are involved in both the first and second parity checks, the values that they were assigned in the first level are used without change in the second level. In general, a digit is assigned a value only at one level in the tree and has the same value in all subsequent levels.

This procedure is continued until all the parity checks have been used to generate levels in the tree. A special case arises if all the digits involved in a parity check have already been assigned values in previous levels. Then the new level is generated by extending only the nodes that satisfy the new parity



First two parity equations:

$$x_2 + x_4 + x_5 = 0$$

$$x_2 + x_7 + x_8 = 0$$

Figure 1.3. The root node and first two levels of a codeword tree for a low-density code.

check. When the procedure is completed, the final level contains a list of all the codewords in the code.

This procedure can be used with any linear block code. In the general case, however, there is no limit on the number of digits involved in a parity check, other than the blocklength n. A typical parity check may involve $n/2$ digits. If this is the first parity check used to generate the codeword tree, there will be $2^{n/2-1}$ nodes in the first level. Even for moderate n, this is too large to be searched by a practical decoder. However, in an (n, j, k) low-density code, each parity check used to define the code involves exactly k digits. The parameter k is typically small, and may be chosen independently of n. All of the low-density codes used in this study have k less than or equal to 8. (Note that this k is not the same as the number of information symbols associated with each codeword, which is also commonly denoted by k.) As will be shown below, codeword trees for low-density codes have fanout limited by 2^{k-1}, and typically the fanout is much less.

One property of the codeword trees obtained using the method given above is the variability of their fanout, or growth rate. This contrasts with codeword trees for convolutional codes, which grow at the same rate at every level. To characterize the growth rate, we first present some definitions. The set of digits involved in a parity check is referred to as a parity check set. Given an ordering of the parity checks, each parity check set is partitioned into two groups, the n-set and the o-set. The n-set, which stands for new set, is made up of the digits not contained in any previous parity check set, and the o-set, or old set, contains the remaining digits. Recall that each level in the codeword tree is associated with a unique parity check. The *set of digits* that are assigned values at a given level in the tree are the digits in the corresponding n-set.

Given a codeword tree for an (n,j,k) low-density code, let n_i be the number of digits in the n-set associated with level i in the tree. Also, let N_i be the total number of nodes at level i, with $N_0 \equiv 1$. Then, for $1 \leq i \leq nj/k$,

(i) If the i^{th} parity check is independent of the preceding parity checks,

$$N_i = 2^{n_i - 1} N_{i-1}. \tag{1.2}$$

(ii) Otherwise,

$$N_i = N_{i-1}. \tag{1.3}$$

In particular,

$$N_i \leq 2^{k-1} N_{i-1}. \tag{1.4}$$

To see why this is true, suppose $n_i \geq 1$. Then there are 2^{n_i} ways to assign values to the n_i digits filled in at level i, and half these assignments will satisfy the corresponding parity check. Since in this case parity check i must be independent of the preceding ones, case (i) applies and (1.2) is valid. On the other hand, suppose $n_i = 0$ and parity check i is independent of the preceding ones. Then, in going from level $i-1$ to level i, the dimension of the set of codeword fragments represented in the tree will decrease by one. Thus

$N_i = N_{i-1}/2$ and (1.2) is again valid. If parity check i is not independent of the preceding ones, then n_i must be zero, thus N_i cannot be greater than N_{i-1}. Furthermore, every codeword fragment satisfying the first $i-1$ parity checks will necessarily satisfy parity check i, thus $N_i = N_{i-1}$. This establishes (1.3). Since every parity check set contains k digits, $n_i \leq k$ for all i. This yields (1.4) and concludes the proof.

This result implies that codeword trees for low-density codes tend to grow quickly near the beginning of the tree and more slowly toward the end. This occurs because the growth rate at a given level depends on the size of the corresponding n-set. At a level close to the origin, few digits will have had a chance to appear in previous parity check sets, hence the corresponding n-set tends to be large. The opposite situation holds near the end of the tree. As a result, n_i will roughly be a decreasing function of i. This is not a hard and fast rule, but a good qualitative description that holds regardless of which parity check ordering is used.

In addition to the variable growth rate, codeword trees for low-density codes and those for convolutional codes differ in another important way. With a low-density code, two different subtrees descending from a common parent node can be identical for many levels. This happens because the set of digit values assigned to a node's children depends only on the parity of the child level's o-set. As a result, two subtrees with a common parent node will agree on the values they assign to codeword digits until a level is reached where the o-set in one tree has different parity than the o-set in the other tree. To determine where such a level can occur, consider the root nodes of the two subtrees. The labels of these two nodes will differ only in the values assigned to the digits in the n-set of the level containing the root nodes. Denote this n-set by S. Then the first level at which the two trees can differ must have an o-set that contains one or more digits in S.

This property can significantly affect the behavior of a sequential decoder. If the decoder diverges from the correct path in the tree, it may not be able to detect its error for many levels. In this case, backtracking one level at a time would be very inefficient. For example, when using a (396,5,6) low-density code and a typical parity check ordering, ten percent of the digits will have 50 or more levels between their appearance in an n-set and their first appearance in an o-set. For this reason, a new sequential decoding algorithm is presented in Section 3.3.

1.3 Properties of Low-Density Codes

This section contains a summary of some properties of low-density codes. They are included here for general interest and because most will be referred to elsewhere in this work. The following results were obtained by Gallager [6], [7] unless stated otherwise.

Recall that a binary (n, j, k) low-density code is a linear block code with blocklength n and a parity matrix with exactly j 1's in each column and k 1's in each row. Note that this k differs from the quantity usually denoted by k in the context of coding – namely, the number of information symbols associated with each codeword. This definition does not define the codes uniquely; there may be many low-density codes for a given value of (n, j, k). Figure 1.4 contains the parity matrix for a (12,3,4) low-density code.

Low-density codes do not exist for all values of n, j, and k. Specifically, the product nj must be a multiple of k. To see why, let L be the number of rows in a low-density code's parity matrix. Counting by rows, there are a total of Lk 1's in the matrix; counting by columns yields a total of nj. Since these expressions must be equal, we have

$$Lk = nj \ , \tag{1.5}$$

and thus nj is a multiple of k.

Furthermore, all the codes considered in this work have n a multiple of k. This restriction is followed by Gallager as well. The blocklength is constrained in this way because it is then quite easy to construct a low-density code for a given set of parameter values. The method of construction is described later in this section. This is not a severe restriction because k is typically small – k is less than 10 for all the codes used in this work.

Equation (1.5) implies that L, the number of rows in the parity matrix, equals nj/k. For this reason, j is always chosen to be less than k. Otherwise L would be greater than n, which is undesirable, because if n of the

$$n=12,\ j=3,\ k=4$$

$$\begin{bmatrix} 0101 & 0010 & 0001 \\ 1000 & 0100 & 1010 \\ 0010 & 1001 & 0100 \\[1ex] 1000 & 1110 & 0000 \\ 0101 & 0000 & 1100 \\ 0010 & 0001 & 0011 \\[1ex] 0001 & 0101 & 0100 \\ 0100 & 1000 & 1001 \\ 1010 & 0010 & 0010 \end{bmatrix}$$

Figure 1.4. The parity matrix of a (12,3,4) low-density code.

parity checks were linearly independent, the code rate would be zero.

Since the rank of the parity matrix equals $n(1-R)$, where R is the code rate in bits, and since the rank of any matrix cannot be greater than the number of rows, we have

$$n(1-R) \leq \frac{nj}{k} . \tag{1.6}$$

Equivalently,

$$R \geq 1 - \frac{j}{k} . \tag{1.7}$$

This relationship is satisfied with equality if and only if all the rows of the parity matrix are linearly independent, and this does not hold for the low-density codes used in this study. However, the parity checks used to define these codes are chosen randomly, and thus one would expect the number of linearly independent parity checks to be close to the total number. For this reason, the quantity $1 - j/k$ is referred to as the designed code rate. In any case, (1.7) remains a valid lower bound.

Gallager studied the minimum distance properties of a random ensemble of these codes. He obtained an upper bound to the minimum distance distribution function $\mathrm{P}(d_{\min} \leq \delta n)$. Note that Gallager's ensemble does not contain all possible low-density codes, but only those obtainable by the construction process described below. For fixed n, j, and k, where n is a multiple of k, the ensemble is defined in the following way. Start with A, the $(n/k) \times n$

matrix shown in Figure 1.5. Row i of A contains 1's in positions $(i-1)k$ through ik and 0's elsewhere. To choose a code from the ensemble, create j new matrices by permuting the columns of A j times. These matrices are created independently, with each possible permutation having equal probability. They are placed together to form a single $(nj/k)\times n$ matrix, the parity matrix of the chosen code. Figure 1.4 contains an example of the parity matrix of a code chosen from the ensemble of (12,3,4) low-density codes. Each submatrix contains exactly k 1's in each row and a single 1 in each column, so the procedure yields a valid (n,j,k) low-density code. Gallager's ensemble differs slightly from the one just described in that the first submatrix is always chosen to be an unpermuted copy of A. However, both ensembles have the same minimum distance distribution function.

Gallager's upper bound to $P(d_{\min}\leq \delta n)$ is fairly complicated and will not be presented (see Theorem 2.4 of [7]). Of interest here is its behavior as n increases, with j and k fixed. When $j\geq 3$, the bound converges to a unit step at δ_{jk}, where δ_{jk} is the only strictly positive zero of the function $B(\lambda)$. This function is given by

$$B(\lambda)=(j-1)h(\lambda)-\frac{j}{k}\Big[\mu(s)+(k-1)\ln 2\Big]+js\lambda\,, \tag{1.8}$$

where

$$h(\lambda)=-\lambda\ln\lambda-(1-\lambda)\ln(1-\lambda), \tag{1.9}$$

$n=12, \ k=4$

$$\begin{bmatrix} 1111 & 0000 & 0000 \\ 0000 & 1111 & 0000 \\ 0000 & 0000 & 1111 \end{bmatrix}$$

Figure 1.5. The matrix A, for $n = 12$ and $k = 4$.

$$\mu(s) = \ln 2^{-k}\left[(1+e^s)^k + (1-e^s)^k\right], \tag{1.10}$$

and s satisfies

$$\mu'(s) = \lambda k \ . \tag{1.11}$$

This means that for large n, almost all codes in the ensemble have minimum distance $d_{\min}$ close to or greater than $\delta_{jk}n$. This linear increase with n of the typical minimum distance is a good property to have and it is uncommon among other classes of codes. In fact, the Justesen codes are the only known class of codes with an explicitly defined construction for which the ratio $\frac{d_{\min}}{n}$ is bounded away from zero, when the rate R is fixed and n goes to infinity (see Section 7.9 of Blahut [2]).

For the case $j = 2$, Gallager showed that

$$d_{\min} \leq 2 + \frac{2\ln\frac{n}{2}}{\ln(k-1)} \ . \tag{1.12}$$

Thus, when $j = 2$ and k is fixed, $\frac{d_{\min}}{n} \rightarrow 0$ as n goes to infinity. For this reason, low-density codes with $j = 2$ are not considered in this work. When $j = 1$, $d_{\min} = 2$, thus codes with this value of j are not considered either.

Another significant property of Gallager's upper bound to $\mathrm{P}(d_{\min} \leq \delta n)$ is that for large n it tends to have a small step at $\delta = 2/n$. The amplitude of this step is $O(n^{-(j-2)})$. This fact led Gallager to construct an expurgated ensemble of low-density codes by throwing out the half with the smallest minimum distance. Otherwise, the codes with $d_{\min} = 2$ dominated his bound on the ensemble average of the decoding error probability when a maximum likelihood decoder was used. In order to avoid codes with $d_{\min} = 2$, all the codes used in this study were constrained to have the following property:

No two parity check sets contain more than one digit in common.

The codes used by Gallager in his simulations were chosen to satisfy this property as well. The term "parity check set" denotes the set of digits involved in a parity check. Note that the parity check sets referred to in the property are those contained in the parity matrix used to define the code – the property does not apply to derived parity checks formed by taking linear combinations of rows of the parity matrix.

This property also ensures the validity of the first step of Gallager's decoding algorithm for low-density codes, described in Section 1.4 of this thesis. The results of the first step are used to generate the reliability information utilized by the decoding algorithms presented in Chapter 3. An algorithm that generates the low-density codes used in this thesis is described in Section A.1.

1.4 Other Decoding Algorithms for Low-Density Codes

This section contains a summary of some known decoding algorithms for low-density codes. The first algorithm is due to Gallager [6], [7]. The first step of this algorithm is incorporated into some of the decoding algorithms presented in Chapter 3. Two algorithms due to Zyablov and Pinsker [9], [10] are included also. One is for the binary erasure channel, and the other is for the binary symmetric channel.

Gallager presented two related decoding algorithms for low-density codes ([6], or pp. 41-45 of [7]). One algorithm can be used only on a binary symmetric channel, that is, a channel with both binary input and binary output. The second algorithm can be used on any binary-input memoryless channel; in particular, it can be used on a channel with real-valued outputs, such as the additive white Gaussian noise channel. The first algorithm is simpler but less powerful than the second algorithm, and only the second algorithm will be described here.

Recall the following terminology. The vector $\mathbf{x} = (x_i)_{i=1}^n$ is the transmitted codeword and $\mathbf{y} = (y_i)_{i=1}^n$ is the channel output, where n is the block-length. In the following discussion, the channel is assumed to be memoryless and able to accept binary inputs, but is otherwise unconstrained. The code being used is a low-density code with parameters (n, j, k). The notation that follows differs from Gallager's.

Gallager's algorithm is iterative. Because it is difficult to describe, a simplified version is presented first. The first iteration starts with the n dimensional vector $\mathbf{z}^{(0)}$, whose components are defined by

$$z_i^{(0)} = \mathrm{P}(x_i = 1 \mid y_i), \quad 1 \le i \le n. \tag{1.13}$$

Note that y_i is the raw channel output, and is not necessarily binary valued. Subsequent values of $\mathbf{z}$ are computed using the following formula.

$$\frac{1-z_i^{(p+1)}}{z_i^{(p+1)}} = \frac{1-z_i^{(p)}}{z_i^{(p)}} \prod_{l=1}^{j} \left[\frac{1+\prod_{m=1}^{k-1}(1-2z_\lambda^{(p)})}{1-\prod_{m=1}^{k-1}(1-2z_\lambda^{(p)})} \right], \quad 1 \le i \le n, \; p \ge 1, \tag{1.14}$$

where $z_\lambda^{(p)}$ is the component of $\mathbf{z}^{(p)}$ corresponding to the m-th digit other than x_i in the l-th parity check containing x_i. The algorithm continues until each component of $\mathbf{z}^{(p)}$ tends toward either 0 or 1. The decoder output is then $(\hat{x}_i)_{i=1}^n$, where $\hat{x}_i$ is the value toward which $z_i^{(p)}$ is converging.

To interpret this procedure, consider the parity check tree shown in Figure 1.6. The root node represents x_i and is said to be in the first level. The nodes in the second level of the tree represent all the digits that are "linked" with x_i, where two digits are linked if they occur in a common parity check. Thus, the j edges leaving the root node represent the j parity checks containing x_i, and each edge leads to $k-1$ nodes. Similarly, nodes in the third level represent digits that are linked with digits in the second level. However, only

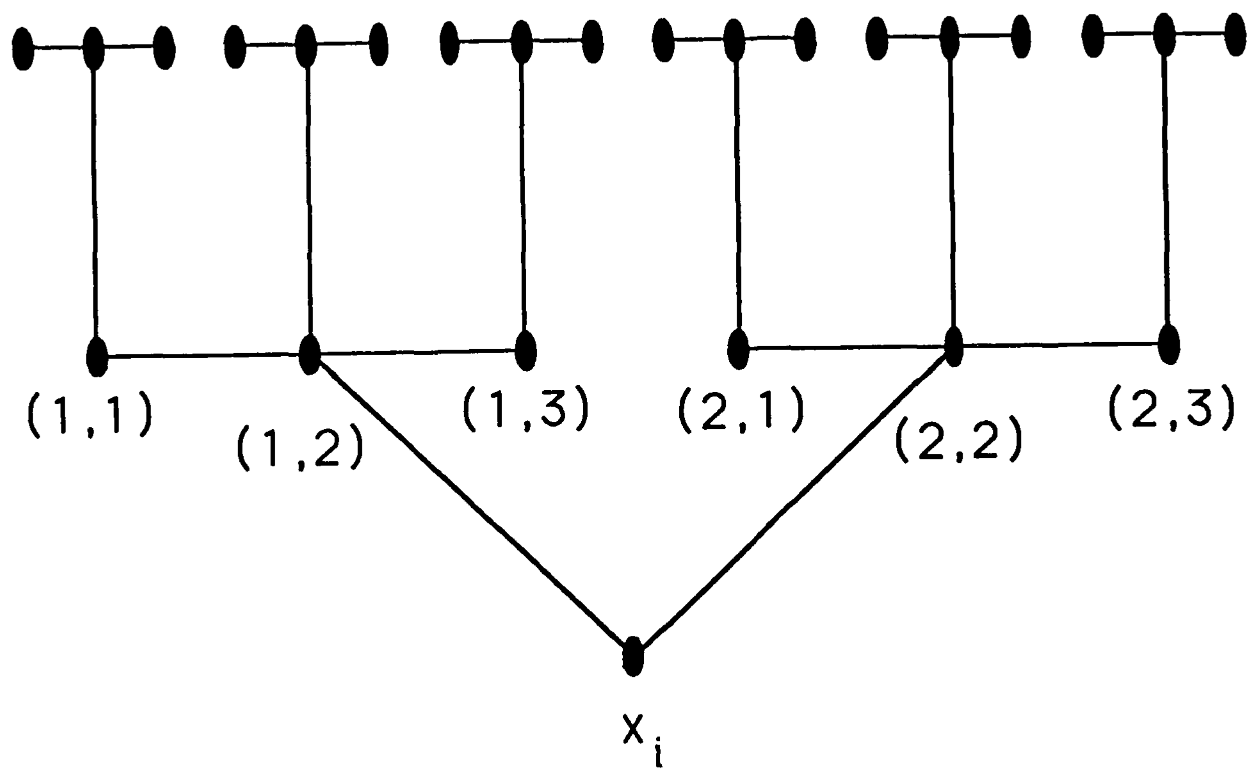


Figure 1.6. The first three levels of the parity check tree rooted at x_i, for $j=2$ and $k=4$. The terms in parentheses refer to (l,m) values used in the update equation.

$j-1$ edges emanate from a node at level two. This happens because for a digit at level two, the parity check containing both it and the root node already has its digits listed in the tree. The parity check tree is extended in this fashion, and continues indefinitely. From the second level on, each node has $j-1$ sets of $k-1$ children. However, note that any given digit will appear more than once in the tree, and when the root node appears again, the tree essentially repeats itself.

Then (1.14) may be interpreted as follows. By definition, $z_i^{(0)}$ is the probability that x_i equals 1, given the channel output y_i. In addition, Gallager showed

$$z_i^{(1)} = \mathrm{P}(x_i = 1 \mid \{y_l, l \in S_i\}) \,, \tag{1.15}$$

where S_i is the set of indices of the digits in the first two levels of the parity check tree rooted at x_i. Thus, from the viewpoint of digit i, $z_i^{(0)}$ equals the probability that x_i equals one, conditioned on the channel output at the first level of the parity check tree, and $z_i^{(1)}$ equals the probability of the same event, conditioned on the channel output at the first two levels of the parity check tree. This may be generalized; the algorithm is designed so that each iteration expands the conditioning set to include one more level of the parity check tree.

However, the algorithm as described above will not achieve this goal. Specifically, the interpretation breaks down for $\mathbf{z}^{(2)}$ and subsequent iterations.

This occurs because a necessary independence assumption about the digits in the conditioning set breaks down. As a concrete example, suppose that for a given low-density code, the parity check tree rooted at digit 15 has digit 20 in the second level. Then, not only will the value of $z_{15}^{(1)}$ depend on $z_{20}^{(0)}$, but $z_{20}^{(1)}$ will depend on $z_{15}^{(0)}$. Therefore, the value of $z_{15}^{(2)}$, which depends on $z_{20}^{(1)}$, will in turn depend on $z_{15}^{(0)}$.

To avoid this problem, Gallager's decoding algorithm differs from the above description in the following way. Each digit i is associated with j probability values, denoted by $z_i^{(p)}(1)$ through $z_i^{(p)}(j)$, instead of the single value $z_i^{(p)}$. Initially, these quantities are the same;

$$z_i^{(0)}(t) = z_i^{(0)} = \mathrm{P}(x_i = 1 \mid y_i), \qquad 1 \le t \le j. \tag{1.16}$$

However, the updates differ. Each $z_i^{(p)}(t)$ ignores one of the j parity checks involving digit i. In other words,

$$\frac{1-z_i^{(p+1)}(t)}{z_i^{(p+1)}(t)} = \frac{1-z_i^{(p)}(t)}{z_i^{(p)}(t)} \prod_{\substack{1 \le l \le j \\ l \ne t}} \left[\frac{1+\prod\limits_{m=1}^{k-1} (1-2z_\lambda^{(p)}(t_\lambda))}{1-\prod\limits_{m=1}^{k-1} (1-2z_\lambda^{(p)}(t_\lambda))} \right], \quad 1 \le i \le n,\ p \ge 1. \tag{1.17}$$

This differs from (1.14) in that the t-th term is omitted in the outermost product. Also, t_λ is chosen so that the update for $z_\lambda^{(p)}(t_\lambda)$ is the one that ignores the parity check containing digit i. The algorithm stops when, for all digits $1 \le i \le n$, all j of the quantities $z_i^{(p)}(1)$ through $z_i^{(p)}(j)$ converge to zero or one. If these quantities do not converge, a decoding failure is declared.

Even with this modification, the independence assumption will be violated at a level in the parity check tree where a digit appears for a second time. As mentioned previously, this must happen eventually. However, Gallager reasons that the dependencies have a minor effect and tend to cancel each other out. Furthermore, after several iterations, the value of $\mathbf{z}^{(p)}$ may be interpreted as the initial value $\mathbf{z}^{(0)}$ corresponding to a received sequence that is easier to decode than the original one. Therefore, the iterations are continued even after dependencies occur.

For this algorithm, Gallager showed that the average number of iterations required to decode is $O(\log \log n)$. He also obtained loose bounds on the probability of decoding error when using a binary symmetric channel. When $j=3$, the probability of decoding error is bounded by some small negative power of n, and when $j>3$, it is bounded by an exponential of a root of n. These bounds are known to be valid only when the crossover probability is sufficiently small; how small it must be depends on j and k. Gallager hypothesized that the probability of decoding error actually decreases exponentially in n.

Apart from its general interest as a decoding algorithm for low-density codes, Gallager's algorithm is described here because of its use in some of the decoding algorithms presented in this thesis. The first iteration of Gallager's algorithm is used to generate a digit reliability measure for some of the binary symmetric channel decoding algorithms presented in Chapter 3. Only a single value is generated for each digit, hence the simplified update Equation (1.14) is used. All the low-density codes used in this study were chosen so that no two parity check sets contain more than one digit in common, thus the independence assumption required to validate the first iteration is satisfied.

Zyablov and Pinsker have also studied low-density codes. They obtained results for the binary erasure channel (BEC) [9] and the binary symmetric channel (BSC) [10]. Both results concern ensembles of codes. For the BEC case, their ensemble contains codes that do not strictly satisfy the definition of low-density codes given in Section 1.1. These codes have a parity matrix with a fixed number of 1's in each column, as before, but the number of 1's in each row is not fixed.

Zyablov and Pinsker define two quantities, α_0 and ω_0, that depend on the number of 1's in each column and the number of rows in the parity matrix. They show that for any $\alpha < \alpha_0$ and $\omega < \omega_0$, with probability P approaching one as n approaches infinity, with other code parameters fixed, the following event occurs. A code from their ensemble has minimum distance $d \geq \omega n$ and has a decoder with complexity $O(n \log n)$ that can correctly decode all erasures with multiplicity $\leq \alpha n$. This decoder complexity is very low for a block code.

A decoder that achieves this result works by identifying the parity checks that involve exactly one erased digit. Each of these erased digits has only one value that will satisfy the associated parity check, thus the decoder is able to fill them in correctly. The process repeats until all erased digits are filled.

Zyablov and Pinsker derive an analogous result for the BSC. In this case, their code ensemble is equivalent to Gallager's. They define a quantity $\alpha_{0.5}$ that depends on the number of 1's in each column and each row of a code's parity matrix. For any $\alpha < \alpha_{0.5}$, with probability P approaching one as n approaches infinity, with other code parameters fixed, a code from their ensemble has a decoder with complexity $O(n \log n)$ that can decode all errors with multiplicity $\leq \alpha n$.

The decoder they use to achieve this bound starts with a fixed partition of the digits into $q = j(k-1)+1$ subsets. The subsets are chosen so that no two digits in a single subset are linked, in the sense defined earlier of occurring together in a common parity check. Since a digit can be linked with at most $j(k-1)$ other digits, such a partition is possible. The decoder considers the q subsets in succession. For each subset, it changes all the digits for which less than half the j parity checks in which they occur are satisfied. The process repeats until at least half the parity checks involving each digit are satisfied. If the resuit is not a valid codeword, a decoding failure is declared; otherwise, the resulting codeword is the decoder output.

CHAPTER 2

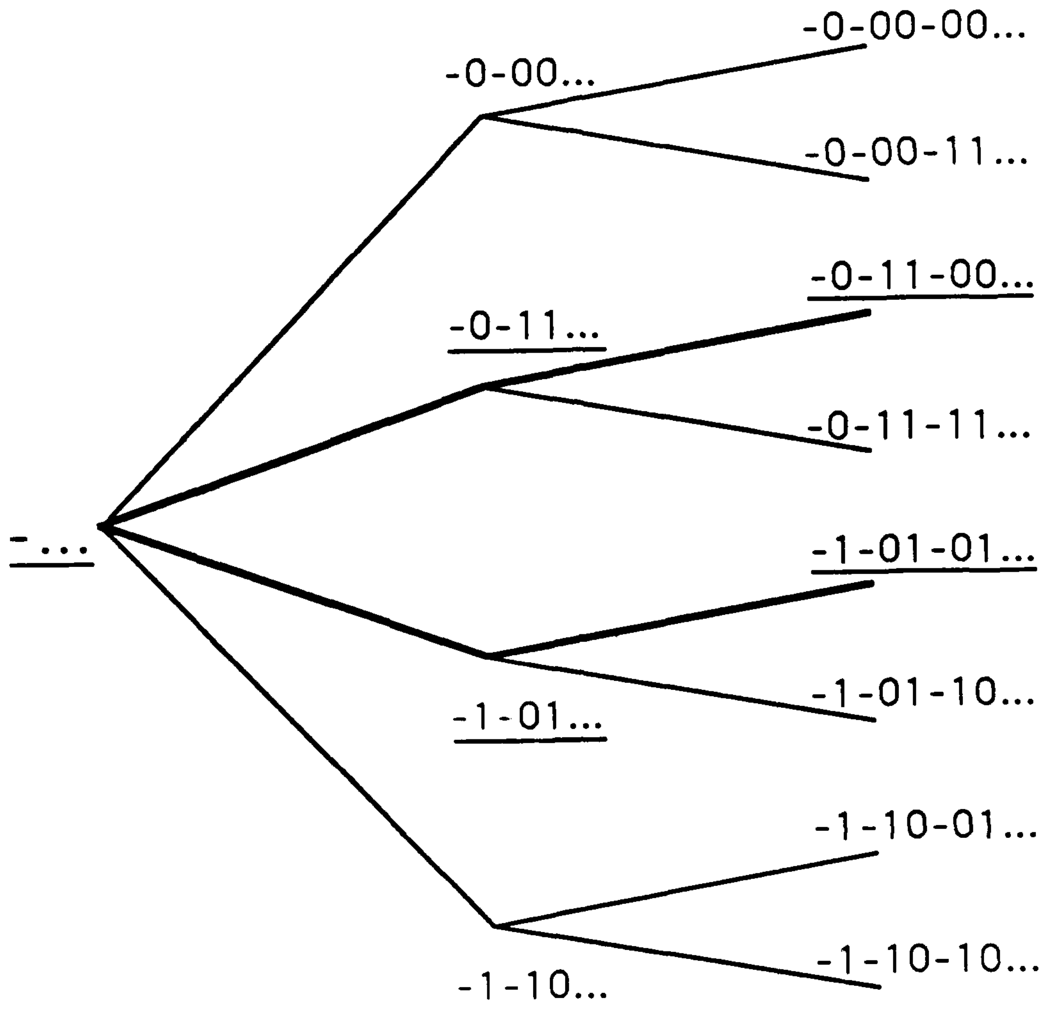
AN SDR ALGORITHM FOR THE BINARY ERASURE CHANNEL

2.1 The MNE Ordering Algorithm

The binary erasure channel (BEC) is a discrete, memoryless channel with transition probabilities as shown in Figure 1.1. The input symbol is erased with probability ϵ, and this probability is independent of the input. If not erased, the input symbol is unchanged.

Recall that an SDR algorithm is composed of two parts – an ordering algorithm, that orders the parity checks, and a sequential decoder. The BEC is well-suited for an ordering algorithm, because the decoder knows exactly where noise occurs in the message. In addition, the corrupted digits are equally unreliable, so an ordering algorithm does not have to compare the effects of different levels of noise. Most important, there exists a tight upper bound to the number of nodes in the codeword tree visited by the sequential decoder, for a given parity check ordering and erasure pattern. Here, the sequential decoder "visits" a node if at some time that node is hypothesized to be on the path corresponding to the transmitted codeword.

This upper bound is found by counting the nodes in the codeword tree that agree with the unerased portion of the received message. (See Figure 2.1.) Let N_i be the number of such nodes at level i in the codeword tree for an (n, j, k) low-density code. Then



$$y = 0e1e10e0\ldots$$

$$N_0 = 1$$
$$N_1 = 2 \quad e_1 = 2 \quad \delta_1 = 0$$
$$N_2 = 2 \quad e_2 = 1 \quad \delta_2 = 0$$

Figure 2.1. An example illustrating the upper bound to the number of nodes visited by the sequential decoder. Nodes that agree with the received message y have underlined labels and darkened edges. This is the same codeword tree fragment shown in Figure 1.3.

$$N_0 = 1 \qquad (2.1)$$
$$N_i = 2^{a_i} N_{i-1} \qquad 1 \leq i \leq M$$

where

$$a_i = e_i + \delta_i - 1 \qquad (2.2)$$

$M = nj/k$

$=$ the number of levels in the codeword tree

$e_i =$ the number of "new" erasures at level i

$=$ the number of erasures in the n-set at level i.

The terms n-set and o-set are defined in Section 1.2. The quantity δ_i is defined as follows. We assign $\delta_i = 1$ if $e_i = 0$ and in addition, either: 1) there are no erasures in the o-set at level i, or 2) the parity of the erasures in the o-set at level i, taken as a group, assumes only one value among the surviving nodes at level i. Otherwise, we assign $\delta_i = 0$. In particular, $\delta_i = 0$ if $e_i \geq 1$, and $\delta_i = 1$ if the parity check at level i is redundant, given the preceding parity checks.

An upper bound to the number of nodes visited by the sequential decoder is given by

$$\overline{N}(\mathbf{y},P) = \sum_{i=0}^{M} N_i = \sum_{i=0}^{M} \prod_{l=1}^{i} 2^{a_l} , \qquad (2.3)$$

where $\mathbf{y}$ is the received message and P is the parity check ordering.

Proof of (2.3): Clearly, if N_i is given by (2.1), then $\overline{N}$ is an upper bound to the number of nodes visited by the sequential decoder. To see why (2.1) is valid, first suppose $e_i \geq 1$. Then there are $2^{e_i - 1}$ ways to fill the e_i erasures and satisfy C_i, the parity check at level i, so $N_i = 2^{e_i - 1} N_{i-1}$.

If $e_i = 0$ and there are no erasures in the o-set at level i, then none of the digits involved in C_i were erased. This means there is exactly one way to extend each of the surviving nodes at level $i-1$, so $N_i = N_{i-1}$.

Now suppose $e_i = 0$ and condition 2) for $\delta_i = 1$ holds. Then, for each surviving node at level $i-1$, the parity of the erased digits in the o-set at level i must match the parity obtained when these erasures are filled in correctly. This implies $N_i = N_{i-1}$.

Finally, if $e_i = 0$ and $\delta_i = 0$, then the parity of the erased digits in the o-set at level i will be both even and odd among different surviving nodes at level $i-1$. By symmetry, the parity for half these nodes will match the parity obtained when the erasures are filled in correctly. Thus, only half the nodes will be extended to level i, so $N_i = N_{i-1}/2$. This establishes (2.1), and therefore (2.3). $\square$

This bound is tight in the sense that for a given erasure pattern and parity check ordering, at least one codeword will, when transmitted, require the sequential decoder to visit $\overline{N}$ nodes.

However, it is not practical to compute $\overline{N}$ because it is difficult to determine when $\delta_i = 1$. Even if one could compute this bound easily, there does not seem to be an efficient way to minimize this bound over the set of possible orderings.

Instead, one can use a greedy heuristic in an attempt to find relatively good orderings. In this method, the ordering is obtained by choosing the parity checks one at a time. The first parity check is chosen to minimize N_1. The second parity check is chosen, from the remaining parity checks, to minimize N_2, and so on. This is done until all the parity checks are chosen, resulting in an ordered list. In general, N_i depends on each of the first i parity checks. However, when N_i is minimized, the first $i-1$ parity checks have already been chosen, so N_i is not minimized over the set of all orderings. Similarly, $\overline{N}$ is not necessarily minimized. However, this formulation leads to an efficient algorithm, and it works well in simulations.

The implemented algorithm, called the Minimum New Erasures (MNE) algorithm, differs from the one described above by ignoring the δ_i term in the formula for a_i (2.2). This is equivalent to choosing C_i by minimizing e_i, hence the algorithm's name. This simplification results in a much more efficient algorithm, and the change in performance is minor. To see this, note that δ_i rarely equals one. Furthermore, ignoring δ_i will make a difference only if a parity check with $\delta_i = 1$ is chosen instead of a parity check with $e_i = 0$ and

$\delta_i = 0$. In any case, all parity checks with $e_i = 0$ will be chosen before any with $e_i \geq 1$.

A straightforward implementation of this algorithm searches the entire set of unused parity checks every time a new parity check is chosen. Also, the number of new erasures involved in each parity check is recalculated for every search, since the definition of "new" changes. For an (n, j, k) low density code, where n is the blocklength, this implementation requires $O(j^2n^2/k)$ computation and $O(jn)$ memory. A more efficient implementation, presented in Section A.2, requires $O(jn)$ computation and $O(jn)$ memory.

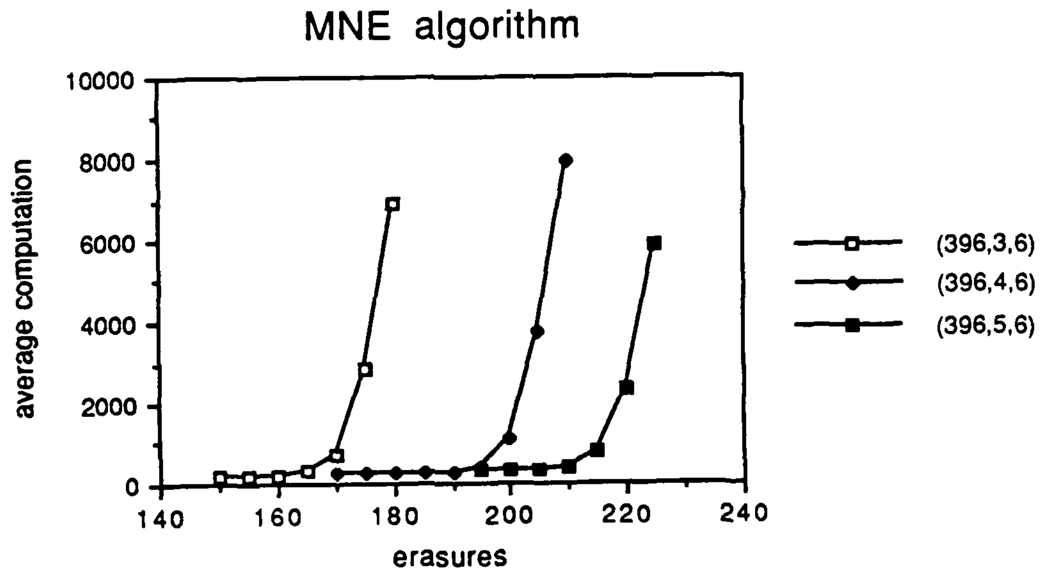
2.2 Simulation Results

The MNE ordering algorithm was implemented on a computer. To form a complete decoder, as discussed in Section 1.1, the MNE algorithm was coupled with the stack algorithm (see Section 7.2.7 of Clark and Cain [3]), a standard sequential decoding algorithm. Although the sequential decoding algorithm described in Section 3.3 was used with the algorithms presented in Chapter 3, it was not used here, because the stack algorithm was found to work sufficiently well for low-density codes on the BEC. It worked well because the stack, which contains only codeword fragments that agree with the unerased portion of the message, stayed small - less than 30 entries for each trial.

Fixed weight pseudorandom erasure patterns were generated and decoded by the computer, and the results were used to estimate the expected amount of computation and the probability of decoding failure. The measure of computation was the number of steps performed by the sequential decoder; specifically, the number of times the entry at the top of the stack was extended. The all-zeros codeword was always used. To compensate for this, the sequential decoder searched the codeword tree branches in reverse numerical order, and thus the all-zeros branch was always searched last. As a result, the expected computation and probability of decoding failure are upper bounds to what would be expected with random codewords.

The MNE algorithm was tested with three low-density codes, with parameters (396,3,6), (396,4,6), and (396,5,6). All three codes have blocklength 396, and their designed rates, given by $1-j/k$, are 1/2, 1/3, and 1/6, respectively. The objectives were to estimate the maximum number of erasures the decoding algorithm could handle and to compare this number with what could theoretically be achieved with standard sequential decoding.

Graphs of average computation and probability of decoding failure versus number of erasures are shown in Figure 2.2. To gauge the effectiveness of the MNE algorithm, simulations were also performed with a random ordering algorithm, which chooses parity check orderings with equal probability for each possible ordering. The results of using the two algorithms with the (396,5,6) low-density code are shown in Figure 2.3. Finally, to extrapolate the



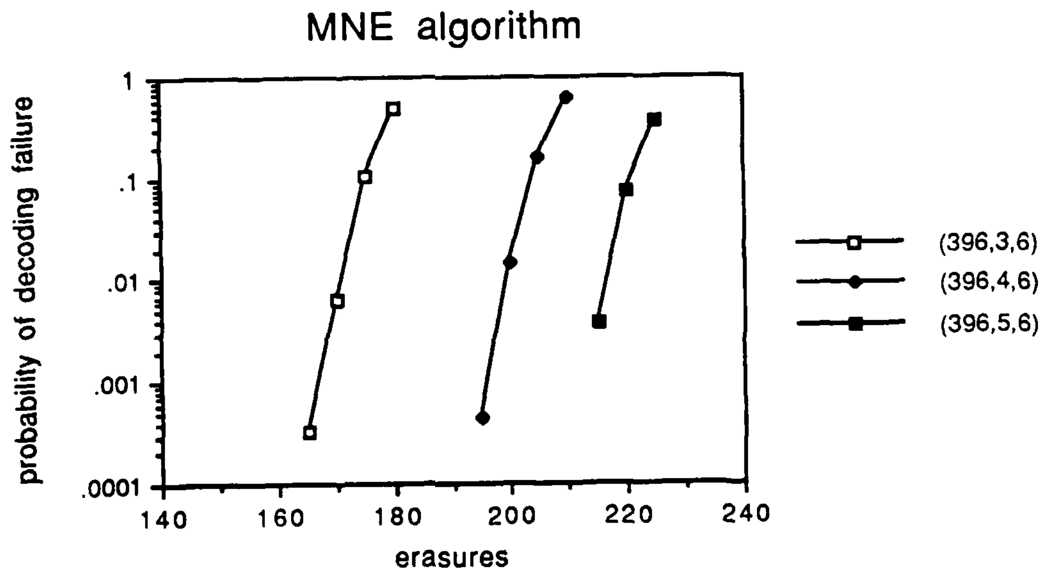
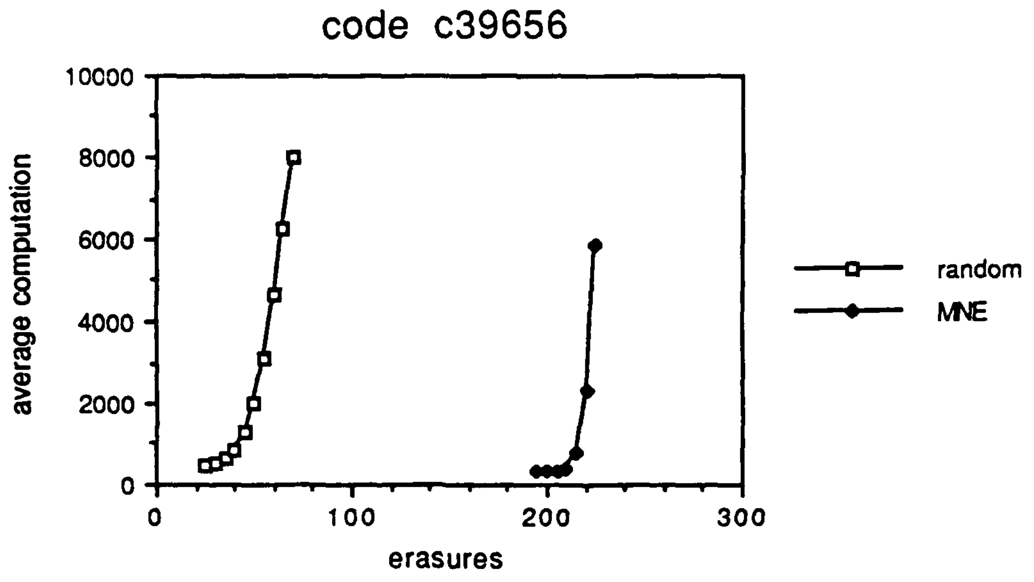


Figure 2.2. Simulation results for the MNE algorithm used with (396,3,6), (396,4,6), and (396,5,6) low-density codes.



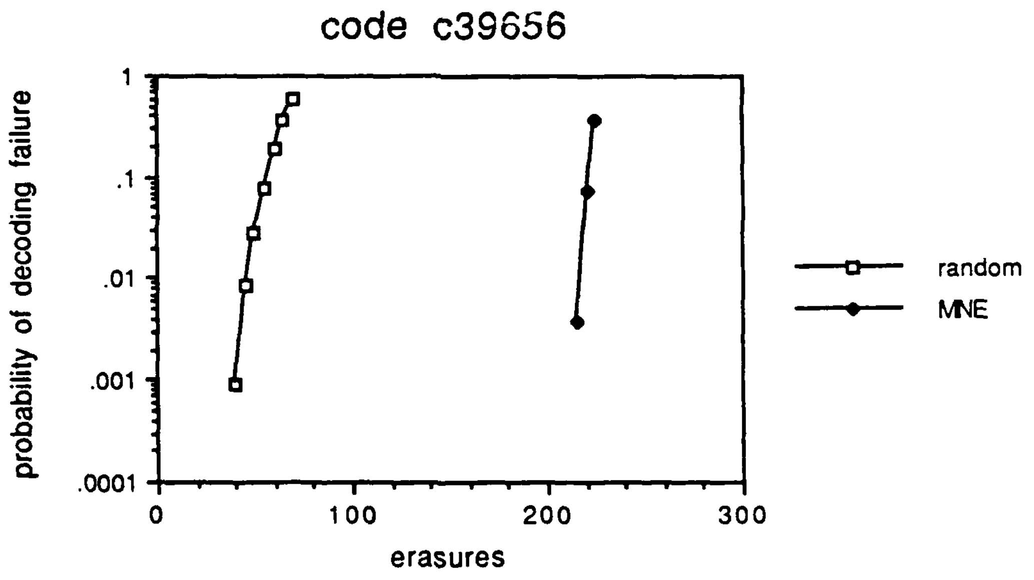
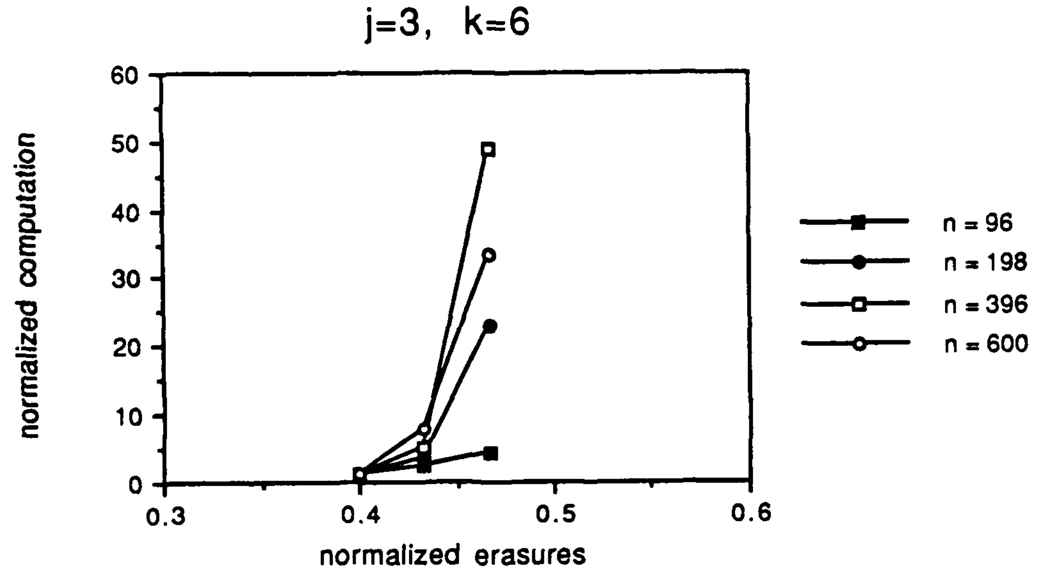


Figure 2.3. Simulation results for random ordering and the MNE algorithm, used with a (396,5,6) low-density code.

results of Figure 2.2 to codes with different blocklengths, simulations were performed using codes with fixed j and k values, but varying blocklengths. This is shown in Figure 2.4, which contains graphs of normalized computation versus normalized erasures for $j=3$, $k=6$ and $j=5$, $k=6$. Here, the normalized erasures are the number of erasures divided by n, and the normalized computation is the amount of computation divided by nj/k, the number of levels in the codeword tree. Note that all codes were chosen to have the property that no two parity check sets contain more than one digit in common, as discussed in Section 1.4. An algorithm to generate such codes is included in Section A.1.

To illustrate how the graphs were obtained, Figure 2.5 contains a table of simulation results for the MNE algorithm used with the (396,5,6) code. The table is broken up into two parts because of space limitations; two sets of trials were not performed for each data point. After each trial, which consisted of decoding a fixed weight psuedorandom erasure pattern, the average computation, $\bar{c}$, and the measured standard deviation of computation, s, were calculated. The standard deviation of $\bar{c}$, denoted by $\hat{s}$, was estimated by $\hat{s} = s/\sqrt{t}$, where t is the number of trials. This relation assumes independent trials. For each data point, the computer performed at least 200 trials and at most 3000; within this range, it stopped if $\hat{s} < (.025)\bar{c}$. The number of trials resulting in decoding errors, N_E, and the number of aborted trials, N_A, were recorded as well. A trial was aborted if the number of steps performed by



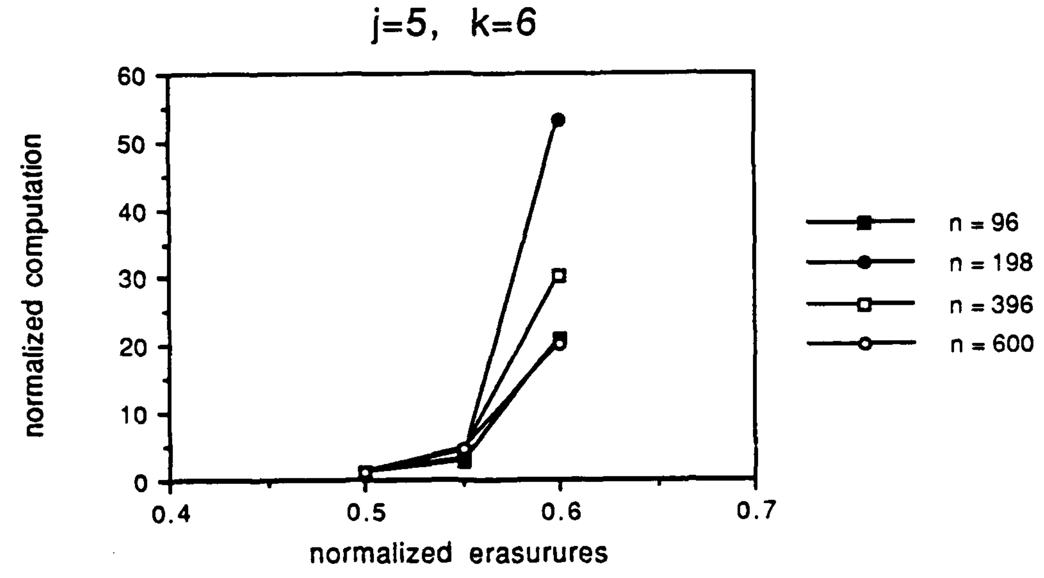


Figure 2.4. Simulation results for the MNE algorithm used with various low-density codes, as shown.

Data point	Erasures e	No. of trials T	Average computation $\bar{c}$	Measured standard deviation s	Standard deviation of estimate $\hat{s}$
1	195	200	331.	0.	0.
2	200	200	331.	0.	0.
3	205	200	332.	10.5	0.74
4	210	225	376.	141.	9.4
5	215	3000	751.	1086.	19.8
6	220	2339	2344.	2832.	58.6
7	225	642	5884.	3724.	147.

Data point	Decoding errors N_E	Aborted trials N_A	Prob. of decoding failure p_{DF}
1	0	0	0.
2	0	0	0.
3	0	0	0.
4	0	0	0.
5	0	11	0.00367
6	0	174	0.0744
7	0	237	0.369

Figure 2.5. Simulation results for the MNE algorithm used with a (396,5,6) low-density code.

the sequential decoder reached ten thousand, or if the stack size reached 200. However, the limit on stack size was unnecessary, because none of the trials aborted for this reason. The estimated probability of decoding failure was given by $p_{DF} = (N_E+N_A)/T$, where T is the total number of trials.

Recall that one objective of the simulations was to estimate e_{MAX}, the maximum number of erasures that could feasibly be decoded by the MNE algorithm. From the results shown in Figure 2.2, one can estimate e_{MAX} to be approximately 165, 195, and 210, respectively, for the (396,3,6), (396,4,6), and (396,5,6) codes. (See Table 2.1.) These are conservative estimates; they are the points at which the average computation first shows a significant increase from its minimum possible value, $nj/k + 1$. From the results shown in Figure 2.3, the maximum number of erasures for random ordering with the (396,5,6) code is approximately 45. As stated above, the performance of the MNE algorithm is significantly better; it can decode roughly 4.7 times as many erasures.

Concerning the probability of decoding failure, note that decoding errors occurred much less often than decoding failures. A decoding error occurs when the sequential decoder completes its search, but outputs the wrong codeword. As defined here, a decoding failure occurs if a trial results in a decoding error, or is aborted due to too much computation. Out of 23,609 trials used to generate the graphs in Figure 2.2, only 4 resulted in decoding errors. All 4 decoding errors occurred using the weakest code – the (396,3,6) code. Thus, if the MNE algorithm is used on a channel with feedback and

Table 2.1. Comparison of the MNE algorithm and standard sequential decoding for the binary erasure channel.

Code parameters			Rate	Achievable e_{MAX} with MNE (approximate)	Max. erasure prob. for sequential decoding	Max. e_{SD} for sequential decoding
n	j	k	R			
396	3	6	1/2	165	0.4142	164.0
396	4	6	1/3	195	0.5874	232.6
396	5	6	1/6	210	0.7818	309.6

retransmission capabilities, one can achieve decoding error rates several orders of magnitude smaller than p_{DF}. However, the effective information rate would be lower than the code rate, due to retransmissions. Note that for the results shown in Figure 2.4, a significant number of decoding errors occurred when using the (96,3,6) and (198,3,6) codes. It seems one should avoid using blocklengths this small, because for the trials used to generate Figure 2.4, none of the other codes had any decoding errors.

What can we conclude about using the MNE algorithm at rates above R_0, the computational cutoff rate for sequential decoding? Recall the estimated values of e_{MAX} for the (396,3,6), (396,4,6), and (396,5,6) codes; they are listed in Table 2.1. These values represent what the MNE algorithm can achieve. It is well-known that the formula for R_0, given in Section 1.1, applied to the BEC yields

$$R_0 = 1 - \log_2(1+\epsilon) \ , \tag{2.4}$$

where ϵ is the channel erasure probability and R_0 is measured in bits per channel use. Inverting this formula yields the maximum erasure probability that sequential decoding can handle for a given code rate. The codes listed above have designed rates of 1/2, 1/3, and 1/6. Corresponding to these rates, the maximum values of ϵ for sequential decoding are listed in Table 2.1. For each code, the resulting expected number of erasures per codeword, given by $n\epsilon$ and denoted by e_{SD}, is listed in Table 2.1 as well.

For the (396,4,6) and (396,5,6) codes, e_{MAX} for the MNE algorithm is significantly less than e_{SD}. Thus, it seems that one cannot exceed R_0 using the MNE algorithm with these two codes. However, the (396,3,6) code looks more promising. The value of e_{SD} for this code's rate and blocklength is 164.0, while e_{MAX} was estimated to be 165. These values are too close to yield a clear conclusion, and when using the MNE algorithm, the maximum acceptable value of ϵ will depend on the specific application. Nevertheless, it is remarkable that a randomly chosen (396,3,6) low-density code performs this well, because the result for standard sequential decoding is an upper bound that applies to all tree codes.

Table 2.1 shows that the MNE algorithm performs better with respect to sequential decoding as the code rate increases. This suggests that the performance may continue to improve with code rates greater than 1/2. However, Gallager's result discussed in Section 1.3 shows that the typical minimum distance of an ensemble of (n,j,k) low-density codes grows linearly in n only if $j \geq 3$. For this reason, simulations were not performed with (396,2,6) or (396,1,6) codes. It would be desirable to study low-density codes with rate $R = 1 - j/k > 1/2$ and $j \geq 3$ to determine whether the MNE algorithm can clearly outperform standard sequential decoding.

To achieve an arbitrarily low probability of decoding error, one must use codes with arbitrarily large blocklengths. This leads one to question whether the results presented above can be extrapolated to low-density codes with larger blocklengths. The graphs shown in Figure 2.4 shed some light on this. Both graphs seem to indicate that the cutoff point, where the average computation starts to increase, occurs at roughly the same value of normalized erasures, as n varies and with j and k fixed. In other words, for fixed j and k, the number of decodable erasures when using the MNE algorithm increases linearly in n. If this hypothesis is true, then using the MNE algorithm with low-density codes is truly comparable to standard sequential decoding, in the sense that codes with arbitrarily long blocklengths will work well on a channel with fixed erasure probability. This behavior is expected, since the typical minimum distance of an ensemble of (n, j, k) low-density codes grows linearly in n, for $j \geq 3$.

Note that this discussion implicitly assumes that the codes used in the simulations are "typical" low-density codes. This assumption is reasonable in light of Gallager's result presented in Section 1.3. It states that almost all the low-density codes in his ensemble have minimum distance greater than a single lower bound, $\delta_{jk}n$. In addition, in Chapter 3, the simulation results obtained using two randomly chosen (396,5,6) low-density codes are found to be very close.

One feature that limits the SDR approach, that is not encountered in standard sequential decoding, is that codeword trees for low-density codes tend to grow more rapidly near the beginning than near the end. This happens regardless of the parity check ordering, as discussed in Section 1.2.

Thus, the early part of the tree looks like a code with higher rate than the original code. This is a drawback, since the sequential decoder works well only at rates less than R_0. However, the MNE algorithm tends to push erased digits toward the end of the tree, and this helps the sequential decoder. The net result of these effects determines whether the overall performance is better than standard sequential decoding. Another limitation of the SDR approach is that the distance properties of low-density codes are probably not as good as those of arbitrary tree codes.

CHAPTER 3

SDR ALGORITHMS FOR THE BINARY SYMMETRIC CHANNEL

3.1 Codeword Tree Ordering Algorithms

The binary symmetric channel (BSC) is a discrete, memoryless channel with transition probabilities as shown in Figure 3.1. With probability p, an error occurs; otherwise, the input symbol is unchanged. The error vector, $\mathbf{e}$, is defined by

$$e_i = x_i \oplus y_i \ , \tag{3.1}$$

where $\oplus$ denotes mod 2 addition. In contrast with the binary erasure channel, it is not immediately apparent when an input symbol is incorrectly received.

Since every digit in the received message, taken by itself, is equally reliable, it would seem that the ordering algorithm has no information to work with. With this view in mind, the algorithm would choose an ordering that in some way minimized the size of the codeword tree, and it would use this ordering regardless of the received message.

However, one may generate a reliability measure for a received digit by considering the values of other digits and using the structure of the code. For example, suppose one of the parity equations of the code states that for every codeword, x_1 and x_5 must sum to zero, mod 2. If y_1 and y_5 are both zero, then the probability that y_1 is in error is less than its a priori probability, p,

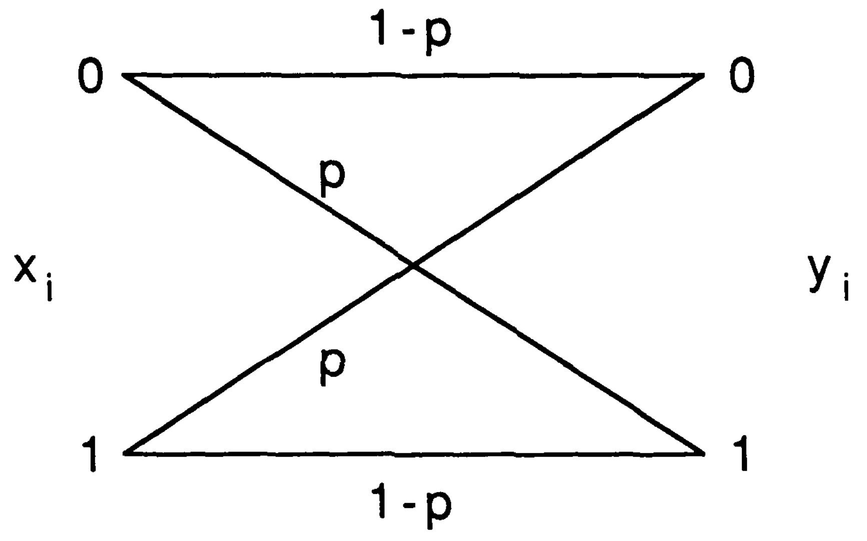


Figure 3.1. The binary symmetric channel.

assuming $p<0.5$ and equally probable codewords. With this approach, it is possible to obtain different reliability levels for different digits, and this information can be used by the ordering algorithm to generate an ordering that depends on the received message.

One may think of a reliability measure for bit i as an estimate of $1-e_i$. In other words, the optimal reliability measure, if it could be computed, would equal one if $y_i = x_i$, and zero otherwise. Since e_i is modeled as a random variable, a Bayesian estimator can be used. One cost function for comparing Bayesian estimators is the mean-squared error , $E(\hat{\theta}-\theta)^2$, where $\hat{\theta}$ is the estimate and θ is the quantity being estimated. Given that we observe **y**, the channel output, the minimum mean-squared error (MMSE) estimate is the conditional mean, $E(1-e_i \mid \mathbf{y})$. Equivalently, this estimate is given by $P(y_i = x_i \mid \mathbf{y})$.

However, it is not advisable to use this estimate in this situation. Conditioning on the entire received message would probably require a great deal of computation, and the ordering algorithm's purpose is to reduce the complexity of the sequential decoder. In addition, computing this estimate corresponds, in a sense, to fully decoding the received message. The output, $\hat{\mathbf{x}}$, would be given by

$$\hat{x}_i = \arg\max_{j=0,1} P(x_i = j \mid \mathbf{y}). \tag{3.2}$$

Instead of being the most probable codeword, $\hat{\mathbf{x}}$ would minimize the probability of error for each digit. Such a scheme, sometimes referred to as probabilistic decoding, does not necessarily output a codeword. This approach would invalidate the proposed structure of the decoding algorithm, because the ordering algorithm is intended only as a preprocessor for the sequential decoder.

Instead, one could consider using a coarse, easily computable reliability measure that would nevertheless lead to an improvement over the static ordering described above. Fortunately, one may compute such a measure for low-density codes. For a given digit i, it is found by considering V_i, the number of violated parity checks involving digit i. The reliability measure is defined by

$$\mu_i = \mathrm{P}(y_i = x_i \,|\, V_i) \,, \tag{3.3}$$

where the joint probability distribution of the x_i's is obtained by assuming each codeword is transmitted with equal probability. This quantity may be interpreted as the MMSE estimate of $1-e_i$, given that we observe only V_i. Assuming that no two parity checks have more than one digit in common, its value is given by the following expression:

$$\mathrm{P}(y_i = x_i \,|\, V_i) = \frac{1}{1 + p\,\beta^{j-2V_i}/(1-p)} \tag{3.4}$$

where

$$\beta = \frac{1-(1-2p)^{k-1}}{1+(1-2p)^{k-1}}$$

$p =$ channel error probability

$j,k =$ low–density code parameters

A proof is given below, but first note that in an (n,j,k) low-density code, each digit appears in j parity checks, thus

$$0 \leq V_i \leq j\ . \tag{3.5}$$

For a given channel and low-density code, $\mathrm{P}(y_i = x_i \mid V_i)$ depends only on V_i, hence the reliability measure will assume only $j+1$ distinct values. These values may be efficiently stored in a table, so that one need not recalculate the above expression.

Proof of (3.4): Let $\mathrm{C}_{i,l}$ be one of the j parity checks involving digit i, A_l the event that $\mathrm{C}_{i,l}$ is not satisfied by **y**, and

$$\alpha = \mathrm{P}(A_l \mid y_i = x_i). \tag{3.6}$$

Parity check $\mathrm{C}_{i,l}$ is not satisfied if and only if there are an odd number of errors among the digits involved in $\mathrm{C}_{i,l}$. There are $k-1$ digits other than i involved in $\mathrm{C}_{i,l}$, and Lemma 4.1 of [7] shows that

$$\begin{aligned}\alpha &= 1 - \frac{1+(1-2p)^{k-1}}{2} \\ &= \frac{1-(1-2p)^{k-1}}{2}\ .\end{aligned} \tag{3.7}$$

Whether or not a parity check is satisfied depends only on the error vector, **e**. Recall the assumption that no two parity checks have more than one digit in common. Then, given $y_i = x_i$, the j events $A_1, \cdots, A_j$ depend on mutually exclusive subsets of **e**, and are therefore independent. Then V_i is binomially distributed with parameters (j, α), and

$$P(V_i = v \mid y_i = x_i) = \binom{j}{v} \alpha^v (1-\alpha)^{j-v}. \tag{3.8}$$

Given $y_i \neq x_i$, A_l occurs if and only if there are an even number of errors over the remaining $k-1$ digits in $C_{i,l}$. Thus,

$$P(A_l \mid y_i \neq x_i) = 1 - \alpha. \tag{3.9}$$

The random variable V_i is again binomially distributed, but now with parameters $(j, 1-\alpha)$.

$$P(V_i = v \mid y_i \neq x_i) = \binom{j}{v} (1-\alpha)^v \alpha^{j-v}. \tag{3.10}$$

The desired probability, $P(y_i = x_i \mid V_i)$, may be written as

$$P(y_i = x_i \mid V_i = v) = \frac{P(V_i = v \mid y_i = x_i)\, P(y_i = x_i)}{P(V_i = v \mid y_i = x_i)\, P(y_i = x_i) + P(V_i = v \mid y_i \neq x_i)\, P(y_i \neq x_i)}. \tag{3.11}$$

Substituting (3.8) and (3.10) into (3.11), and defining $\beta = \alpha/(1-\alpha)$, we obtain (3.4). □

This reliability measure is closely related to one of Gallager's decoding algorithms for low-density codes (pp. 42-45 of Gallager [7]). A brief description of the algorithm is given here; it is described more fully in Section 1.4. Each digit is associated with j values between zero and one, and these values are interpreted as the probability that the transmitted symbol was a one, conditioned on different subsets of the received message. The algorithm is iterative. Initially, each probability is conditioned on the received value of a single digit, and in each iteration the conditioning sets are expanded to include more of the received message. If the decoder is successful, all the values associated with each digit converge either to zero or to one. The independence assumptions used to derive the update equations are violated after a relatively small number of iterations, but the procedure is continued anyway and is found to yield a good decoding algorithm.

The results of the first iteration of Gallager's algorithm are very similar to the reliability measure defined above. Let i be a given digit. In the first iteration, the conditioning sets for i are expanded to include S_i, the set of all digits that occur in the j parity checks containing i. However, to validate the next iteration, each of the j probability values associated with i ignores the digits involved in one of the parity checks. If, instead, we calculate a single value that considers all the digits in S_i, then we obtain the same quantity as the reliability measure defined above, under the conditions defined below.

Recall that our reliability measure is not defined by conditioning on S_i, but on V_i, the number of violated parity checks containing i. Even though V_i is a coarser statistic than S_i, the two definitions result in the same value if the following two conditions hold:

(1) Each codeword in the code is transmitted with equal probability.

(2) There is no derived parity equation, involving only digits in S_i, that is satisfied by the code and linearly independent of the j equations involving digit i.

This is proved as Proposition 3.1 below. We use V_i instead of S_i because (3.4) is easier to evaluate, though less general, since it applies only to the BSC, than the formula for $\mathrm{P}(y_i = x_i \,|\, S_i)$ given by Gallager. For comparison, $\mathrm{P}(y_i = x_i \,|\, S_i)$ is given by any one of Equations (4.1) or (4.6) of [7], or Equation (1.14) in this thesis. To show that $\mathrm{P}(y_i = x_i \,|\, V_i) = \mathrm{P}(y_i = x_i \,|\, S_i)$, we can obtain (3.4) directly from Gallager's formula for $\mathrm{P}(y_i = x_i \,|\, S_i)$, by inserting values appropriate for the BSC. However, Gallager's probability model is defined differently than the one used here. Nevertheless, we can show that the models are equivalent when we restrict our attention to the digits in S_i, which is sufficient for our purposes. Instead, we will show $\mathrm{P}(y_i = x_i \,|\, V_i) = \mathrm{P}(y_i = x_i \,|\, S_i)$ for the model used here - namely, assumptions (1) and (2) above, together with a memoryless BSC.

Note that an equation as described in (2) can occur because the codewords satisfy not only the nj/k parity checks in the low-density code's definition, but all linear combinations of them as well. However, as Gallager states, the effect of the extra dependencies that occur if condition (2) is violated is typically small.

We use the following fact. Its proof is elementary and is omitted here.

Fact. Let C be a linear code with blocklength n, and let $R \subseteq \{1,2,\ldots,n\}$. Let $PC(R)$ be the set of all possible parity check equations that involve only digits in R and that are satisfied by all codewords in C, and let x_R be an assignment of values to digits in R. Then x_R will match some codeword in C if and only if x_R satisfies $PC(R)$. In addition, if a random codeword is chosen uniformly from C, then every valid x_R is equally probable.

Proposition 3.1. $P(y_i = x_i \mid V_i) = P(y_i = x_i \mid S_i)$, under conditions (1) and (2) stated above.

Proof: Let $R \subseteq \{1,2,\ldots,n\}$ be the union of the j parity check sets involving digit i. In other words, R is the set of indices of the digits that make up S_i. For all vectors in this proof, we restrict our attention to elements indexed by R. This applies to codewords as well as error vectors.

Define $v(\mathbf{a})$ for a vector $\mathbf{a} \in \{0,1\}^R$ to be the number of violated parity checks involving digit i. Suppose $\mathbf{a}_1$ and $\mathbf{a}_2$ satisfy $\nu(\mathbf{a}_1)=\nu(\mathbf{a}_2)$. Then we will show

$$P(y_i=x_i \,|\, S_i=\mathbf{a}_1)=P(y_i=x_i \,|\, S_i=\mathbf{a}_2)\,. \tag{3.12}$$

Let $T_1=\{(\mathbf{x}_1^{(1)},\mathbf{e}_1^{(1)}),(\mathbf{x}_1^{(2)},\mathbf{e}_1^{(2)}),\ldots\}$ be the set of all pairs of codewords and error vectors that result in a received message with $S_i=\mathbf{a}_1$, and similarly define $T_2=\{(\mathbf{x}_2^{(1)},\mathbf{e}_2^{(1)}),(\mathbf{x}_2^{(2)},\mathbf{e}_2^{(2)}),\ldots\}$ for $\mathbf{a}_2$. We will demonstrate a one-to-one mapping from T_1 onto T_2, and use this to show $P(T_1)=P(T_2)$. The underlying probability model is that the transmitted codeword is chosen uniformly from the set of all codewords, and the error vector is generated by a memoryless BSC.

Note that every valid codeword occurs in both T_1 and T_2. Recall that we consider only digits with indices in R. From the assumptions and the Fact, a valid codeword is a vector in $\{0,1\}^R$ that satisfies the j parity checks that involve digit i. In addition, each such vector is equally probable.

Without loss of generality, we can assume that the violated parity checks that contribute to $\nu(\mathbf{a}_1)$ are the same parity checks that contribute to $\nu(\mathbf{a}_2)$. This implies that $\mathbf{x}+\mathbf{a}_1+\mathbf{a}_2$ is a valid codeword if $\mathbf{x}$ is. Otherwise, we can achieve this condition by rearranging the digits in each of the vectors in T_2, creating a new set T_2'. Specifically, the digits involved in some violated parity checks are interchanged with those involved in some satisfied checks. The codewords are still valid, and clearly $P(T_2')=P(T_2)$.

The desired mapping ϕ from T_1 to T_2 is given by

$$\phi((\mathbf{x},\mathbf{e})) = (\mathbf{x}+\mathbf{a}_1+\mathbf{a}_2,\mathbf{e}) . \tag{3.13}$$

It is clearly one-to-one and onto. Since each codeword is equally probable, this shows $P(T_1) = P(T_2)$. Equivalently, since $P(T_1) = P(S_i{=}\mathbf{a}_1)$ and $P(T_2) = P(S_i{=}\mathbf{a}_2)$, we have $P(S_i{=}\mathbf{a}_1) = P(S_i{=}\mathbf{a}_2)$. Similarly, we can show $P(y_i{=}x_i$ and $S_i{=}\mathbf{a}_1) = P(y_i{=}x_i$ and $S_i{=}\mathbf{a}_2)$. In this case, the error vectors in T_1 and T_2 are constrained to be zero in position i. Therefore,

$$P(y_i{=}x_i\,|\,S_i{=}\mathbf{a}_1) = P(y_i{=}x_i\,|\,S_i{=}\mathbf{a}_2) . \tag{3.14}$$

Finally, since $\mathrm{P}(y_i = x_i\,|\,V_i)$ is a weighted average of $P(y_i{=}x_i\,|\,S_i{=}\mathbf{a})$ for all $\mathbf{a}$ with $\nu(\mathbf{a}){=}V_i$, this implies $\mathrm{P}(y_i = x_i\,|\,V_i) = \mathrm{P}(y_i = x_i\,|\,S_i)$. $\square$

Once we have an easily computable reliability measure, we must decide how to incorporate it in an ordering algorithm. This question may be addressed by considering the ordering algorithm for the binary erasure channel (BEC). To recall, the BEC algorithm does not compare complete parity check orderings; instead, it constructs a single ordering by choosing parity checks one at a time. At each stage, the chosen parity check is one that minimizes the value of a certain function (the objective function), and this function is derived from an upper bound to the number of nodes visited by the sequential decoder.

This approach is not directly applicable to the BSC; unlike the case with the BEC, we have not found a reasonably tight bound on the number of nodes visited by the sequential decoder, given the received message and the parity check ordering. It seems that such a bound would have to take into account the fine structure of the low-density code being used (to a greater extent than the BEC bound, which only considered the overlap between a given parity check and the union of the previously chosen parity checks).

Nevertheless, one can obtain an effective algorithm by choosing an appropriate objective function. Specifically, we consider objective functions that tend to maximize the reliabilities of the new digits and tend to minimize the number of new digits, where the new digits in a parity check are the ones not contained in any of the previously chosen parity checks. Recall that if a given parity check is chosen to be ith in the final ordering, then the new digits in this parity check are the ones filled in at level i in the codeword tree.

Consider the effect of this algorithm on a given level of the codeword tree. If we maximize the reliability of the new digits, then the sequential decoder will be less likely to start along an incorrect path, or if it is already on an incorrect path, it will be more likely to detect this. On the other hand, when we minimize the number of new digits, the growth of the tree at this level will also be minimized. As a result, there are fewer nodes to search, and there is less likely to be an outgoing branch close enough to the correct branch to appear correct. However, it may not be possible to fulfill these two aims simultaneously, thus the objective function will have to define their relative importance.

Since the algorithm avoids choosing parity checks with unreliable digits, the overall effect is to push noisy digits deeper into the codeword tree. To see why this is a good thing to do, recall that regardless of the parity check ordering, the codeword tree tends to grow more slowly at deeper levels. (See Section 1.4.) For this reason, a wrong move by the sequential decoder at the beginning of the tree will probably require more computation to resolve than a wrong move toward the end. In addition, errors toward the end of the tree are less likely to cause a wrong move, especially if the error occurs at a level where the tree does not grow in size (one new digit per level).

These objective functions do not depend on the reliabilities of the old (i.e., already filled) digits, but there are situations where such dependence may be desirable. For example, one could choose successive parity checks to minimize the reliabilities of the old digits in the new parity checks in order to detect incorrect moves by the sequential decoder sooner, since an incorrect assignment can be detected only at a level where it occurs as an old digit. However, it would be difficult to tell when this would be helpful, since an incorrect assignment of an old digit may be interpreted as a transmission error in a new digit. In any case, it seems that this would rarely be more important than the factors discussed above.

Several different objective functions with the properties described above were used in algorithms implemented on a computer. They are listed in Figure 3.2. A description of these functions is given below, and simulation results are presented in Section 3.2. First, we present some definitions, and discuss the computational complexity of the algorithms. We use the following terminology. The objective function is denoted by E, and its domain is the set of parity checks. For a given parity check, N is the set of new digits, which are defined as above.

Each ordering algorithm operates as follows. The objective function is evaluated at each parity check, and a parity check that minimizes the function is chosen. Since N will change for some of the parity checks, E is re-evaluated for each remaining parity check before the next choice is made. This procedure is repeated until a complete ordering is obtained.

The approach described above requires $O(j^2n^2/k)$ computation and $O(jn)$ memory. A more efficient implementation is possible if E assumes only a finite set of values, and has the additional property that when a parity check's E value changes, it cannot return to a previously held value. These conditions are satisfied by discretized versions of the objective functions described below, except for functions 7.1 and 7.2. With these conditions, an implementation exists with $O(nj+njV/k)$ computation and $O(nj+njV/k)$ memory, where V is the number of values E can assume. The implementation is similar to the $O(n)$ version of the MNE algorithm presented in Section

1. $$E = \sum_{i \in N} (\alpha V_i + \beta)$$

1.1 $\alpha=1,\ \beta=0$

1.2 $\alpha=1,\ 0<\beta<1/k$

1.3 $\alpha=0,\ \beta=1$

1.4 $0<\alpha<1/jk,\ \beta=1$

2. $$E = \begin{cases} \sum_{i \in N} V_i / |N| & \text{if } |N| \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

3. $$E = \sum_{i \in N} h(p_i)\ , \quad h(x) = -x \log x - (1-x)\log(1-x)$$

4. $$E = \prod_{i \in N} [1+2\sqrt{p_i(1-p_i)}\,]$$

5.1 $$E = 2^{|N|-1} \sum_{i \in N} \min(p_i, 1-p_i)$$

5.2 $$E = \sum_{i \in N} \min(p_i, 1-p_i)$$

6.1 $$E = 2^{|N|-1}[1-\prod_{i \in N} \max(p_i, 1-p_i)]$$

6.2 $$E = 1-\prod_{i \in N} \max(p_i, 1-p_i)$$

7.1

$$E = \sum_{\substack{0 \leq l \leq j, \\ total(l) \neq 0}} \frac{[actual(l)-desired(l)]^2}{(total(l))^2}$$

7.2

$$E = \sum_{\substack{0 \leq l \leq j, \\ total(l) \neq 0}} \frac{|actual(l)-desired(l)|}{total(l)}$$

Figure 3.2. List of objective functions.

A.2. The main difference from the MNE algorithm concerns the value of each digit's contribution to E. With the MNE algorithm, E equals the number of erasures in N. Thus, a digit's E contribution is one or zero, depending on whether or not the digit is an erasure. With the BSC objective functions, we must store each digit's E contribution in a size n array. Using this array, we can determine the effect on E of removing a digit from N in a constant number of steps. Without the array, we must recalculate E from scratch, which requires $O(k)$ steps.

3.1.1 Objective functions

Definitions. As above, E is the objective function, and N is the set of new digits for a parity check. For $1 \leq i \leq n$, we define

$$p_i = P(y_i \neq x_i \mid V_i) . \tag{3.15}$$

The function P_0 is a probability distribution on $\mathbf{x}$ and $\mathbf{y}$ that differs from the BSC model by setting $\mathrm{P}_0(y_i \neq x_i) = p_i$.

1. $$E = \sum_{i \in N} (\alpha V_i + \beta) ,$$

where α and β are constants, and V_i is the number of violated parity checks involving y_i. Using V_i itself as a quantitative measure of uncertainty was arbitrary, but this objective function was found to work well. Different values of α and β yield different orderings, and the following values were implemented.

1.1 $\alpha=1,\ \beta=0$

Minimizes $\sum_{i\in N} V_i$.

1.2 $\alpha=1,\ 0<\beta<1/k$

Since each V_i is an integer, and $|N|\leq k$, this causes the algorithm to minimize $\sum_{i\in N} V_i$, and break ties by minimizing $|N|$.

1.3 $\alpha=0,\ \beta=1$

This yields a static ordering, which minimizes $|N|$. The ordering is independent of the received vector **y**. It is obtained without any knowledge of the reliabilities of the individual digits.

1.4 $0<\alpha<1/jk,\ \beta=1$

Since $V_i\leq j$, $\sum_{i\in N} V_i\leq jk$. Thus, the algorithm minimizes $|N|$ and breaks ties by minimizing $\sum_{i\in N} V_i$.

2. $$E=\begin{cases}\sum_{i\in N} V_i/|N| & \text{if } |N|\geq 1,\\ 0 & \text{otherwise.}\end{cases}$$

This algorithm tries to choose a parity check with smallest uncertainty per new digit.

3. $$E = \sum_{i \in N} h(p_i),$$

where h is the entropy function,

$$h(x) = -x \log x - (1-x)\log(1-x) .$$

4. $$E = \prod_{i \in N} [1+2\sqrt{p_i(1-p_i)}\,]$$

One interpretation for this algorithm is that it maximizes $\sum_{i \in N} (R_0(p_i) - r_i)$,

where

$$\begin{aligned} R_0(p_i) &= 1 - \log_2[1 + 2\sqrt{p_i(1-p_i)}\,] \\ &= \text{the computational cutoff rate in bits for} \\ &\quad \text{a BSC with crossover probability } p_i \end{aligned}$$

and

$$\begin{aligned} r_i &= (|N|-1)/|N| \\ &= \text{the approximate local code rate in bits.} \end{aligned}$$

Motivation for this objective function comes from two bounds to the expected amount of computation performed by a sequential decoder – an upper bound due to Gallager (p. 279 of [5]) and a lower bound due to Arikan (Lemmas 3.1 and 6.1 of [1]). If one extends these bounds to the case where the code rate and the crossover probability can vary with time, the given objective function minimizes both these bounds. However, the original bounds are loose, and the extensions may not be valid.

5.1 $$E = 2^{|N|-1} \sum_{i \in N} \min(p_i, 1-p_i)$$

5.2 $$E = \sum_{i \in N} \min(p_i, 1-p_i)$$

The expression $\sum_{i \in N} \min(p_i, 1-p_i)$ is a union bound approximation to $P_0(\tilde{x}_i \neq x_i \text{ for some } i \in N)$, where P_0 is the probability distribution defined previously, $\tilde{x}_i = \arg\max_{x=0,1} \Gamma(x, y_i)$, and Γ is the sequential decoder metric. Since $\Gamma(\tilde{x}_i, y_i) > \Gamma(1-\tilde{x}_i, y_i)$, the sequential decoder will most likely assume $x_i = \tilde{x}_i$ on its first pass. The quantity $2^{|N|-1}$ is a cost chosen to reflect the local growth of the codeword tree if the current parity check were chosen.

6.1 $$E = 2^{|N|-1} [1 - \prod_{i \in N} \max(p_i, 1-p_i)]$$

6.2 $$E = [1 - \prod_{i \in N} \max(p_i, 1-p_i)]$$

The expression $[1 - \prod_{i \in N} \max(p_i, 1-p_i)]$ equals $P_0(\tilde{x}_i \neq x_i \text{ for some } i \in N)$. As with objective function 5.1, $2^{|N|-1}$ is a cost associated with this event.

7.1 $$E = \sum_{\substack{0 \le l \le j, \\ total(l) \neq 0}} \frac{[actual(l) - desired(l)]^2}{(total(l))^2}$$

7.2 $$E = \sum_{\substack{0 \le l \le j, \\ total(l) \neq 0}} \frac{|actual(l) - desired(l)|}{total(l)}$$

Here, *total*(l) is the total number of digits in the received message with

$V_i = l$. The quantity $desired(l)$ is defined to be $(n_0/n)total(l)$, where n_0 is the number of digits encountered in the codeword tree so far, and $actual(l)$ is the number of such digits with $V_i = l$. Thus, these objective functions try to distribute the digits with a given value of V_i evenly throughout the tree.

3.2 Simulation Results

The objective functions described in Section 3.1 were used in codeword tree ordering algorithms implemented on a computer. To form a complete decoder, as discussed in Section 1.1, each ordering algorithm was coupled with a sequential decoder. Because standard sequential decoding algorithms did not work well with low-density codes on the BSC, a new sequential decoder was used. It is described in Section 3.3.

The simulations performed for the BSC algorithms are similar to those described in Section 2.2. Fixed weight psuedorandom error vectors were generated and decoded by a computer, and the results were used to estimate the expected amount of computation and the probability of decoding failure for each algorithm. In this case, the measure of computation was the number of forward and lateral moves performed by the sequential decoder. As before, the all-zeros codeword was always used. To compensate for this, the sequential decoder searched the codeword tree branches in reverse numerical order, thus the all-zeros branch was always searched last. However, because of the sequential decoder used, the expected computation and probability of decoding failure are not necessarily upper bounds to what would be obtained with random codewords, as was the case with the BEC results.

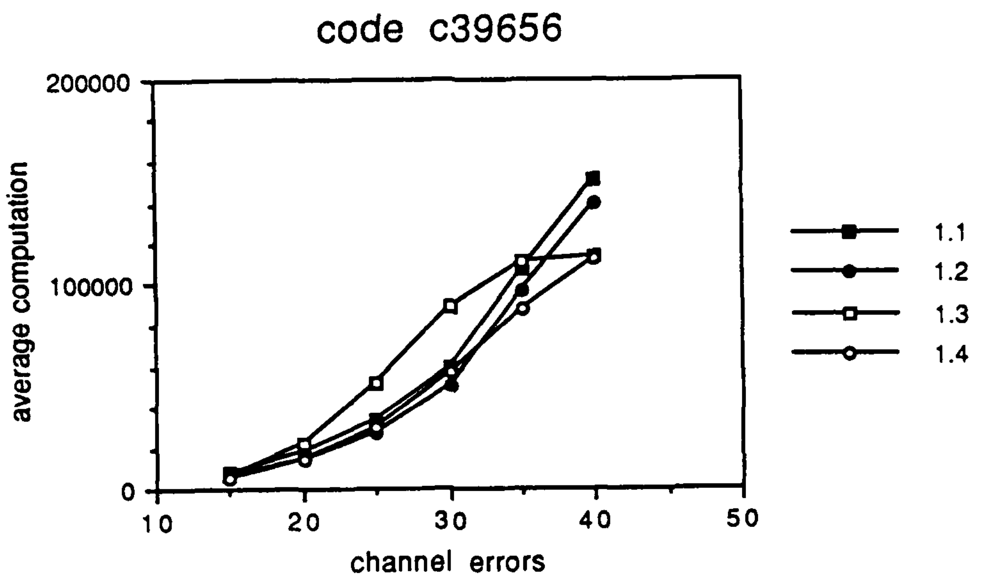
The objective functions were compared using a fixed (396,5,6) low-density code, denoted by c39656. Simulations were performed to estimate the expected computation and probability of decoding failure as a function of the number of channel errors. To study the effects of different j and k values, objective functions 1.2 and 3 were tested with (396,3,6) and (400,4,8) low-density codes as well. To gauge the effectiveness of the algorithms, we implemented a random ordering algorithm, that chooses parity check orderings with equal probability for each, with the (396,5,6) and (396,3,6) codes. Finally, to investigate how typical c39656 was compared to other (396,5,6) codes, objective functions 1.1 and 3 were used with b39656, another (396,5,6) low-density code. Graphs of the results are shown in Figures 3.4 through 3.10. Note that all codes were chosen to have the property that no two parity check sets contain more than one digit in common, as discussed in Section 1.4. A computer program to generate such codes is included in Section A.1.

To illustrate how the graphs were obtained, Figure 3.3 contains a table of simulation results for objective function 1.1 used with code c39656. Each data point was generated using a procedure similar to that used in Section 2.2; the differences are described below. The quantities $\bar{c}$, s, and $\hat{s}$ are the same as before, and the same stopping criterion was used. However, the sequential decoder used here can declare a decoding failure, unlike the stack algorithm

Data point	Channel errors e	No. of trials T	Average computation $\bar{c}$	Measured standard deviation s	Standard deviation of estimate $\hat{s}$
1	15	1442	5965.	5660.	149.
2	20	670	14750.	9533.	368.
3	25	857	26948.	19701.	673.
4	30	810	50208.	35719.	1255.
5	35	535	96797.	55968.	2420.
6	40	219	139605.	51523.	3482.

Data point	Decoding failures N_F	Decoding errors N_E	Aborted trials N_A	Prob. of decoding failure p_{DF}
1	0	0	0	0.
2	0	0	0	0.
3	15	0	0	0.018
4	70	0	0	0.086
5	187	0	0	0.34
6	164	0	0	0.75

Figure 3.3. Simulation results for objective function 1.1 used with code c39656. Additional trials were performed to obtain better estimates of p_{DF} for $e = 15$ and $e = 20$. For $e = 15$, there were zero decoding failures out of 4000 trials, and for $e = 20$, there were 3 decoding failures out of 4000 trials.



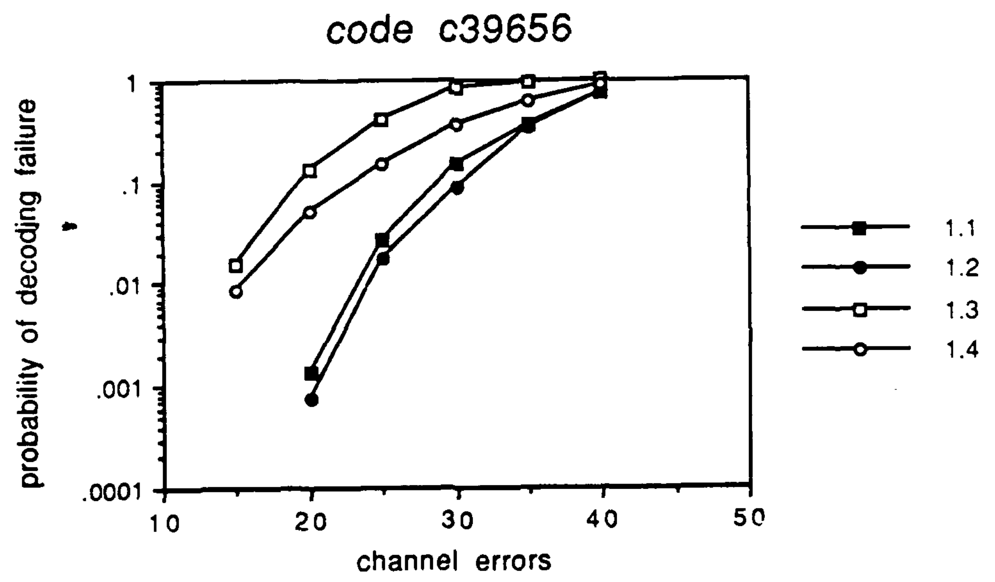
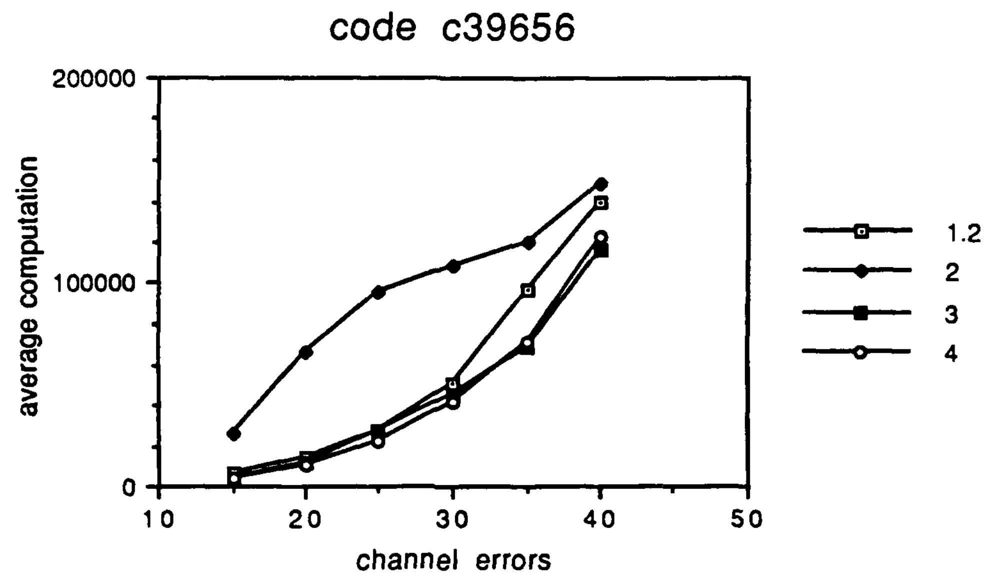


Figure 3.4. Simulation results for objective functions 1.1, 1.2, 1.3, and 1.4 used with a (396,5,6) low-density code. For both objective functions 1.1 and 1.2, with 25 channel errors, there were zero decoding failures out of 4000 trials.



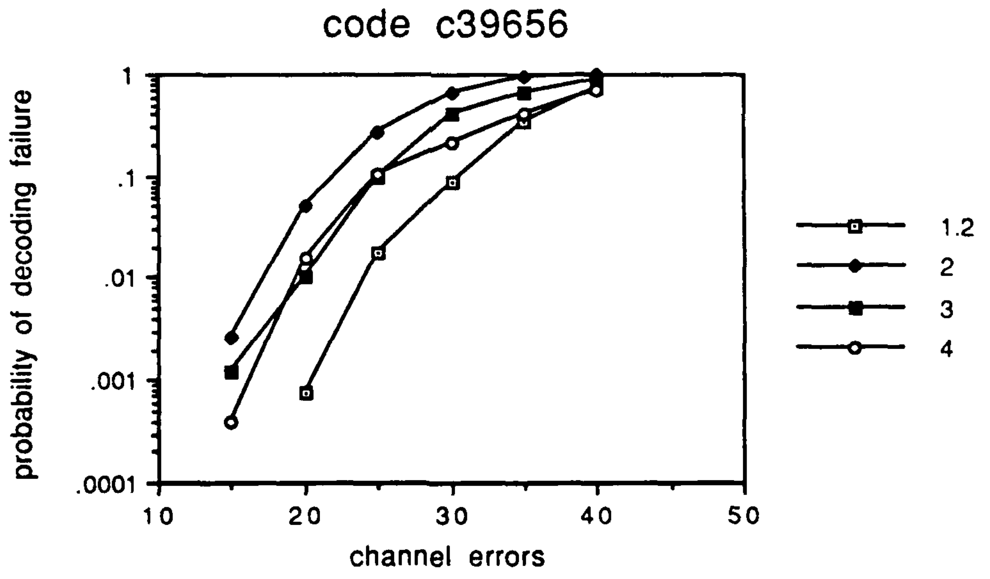
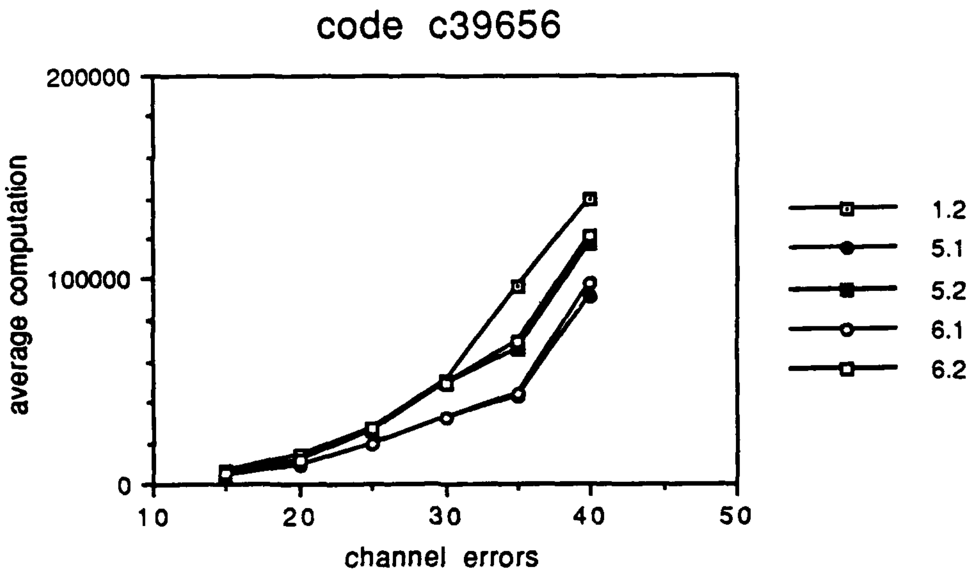


Figure 3.5. Simulation results for objective functions 1.2, 2, 3, and 4 used with a (396,5,6) low-density code.



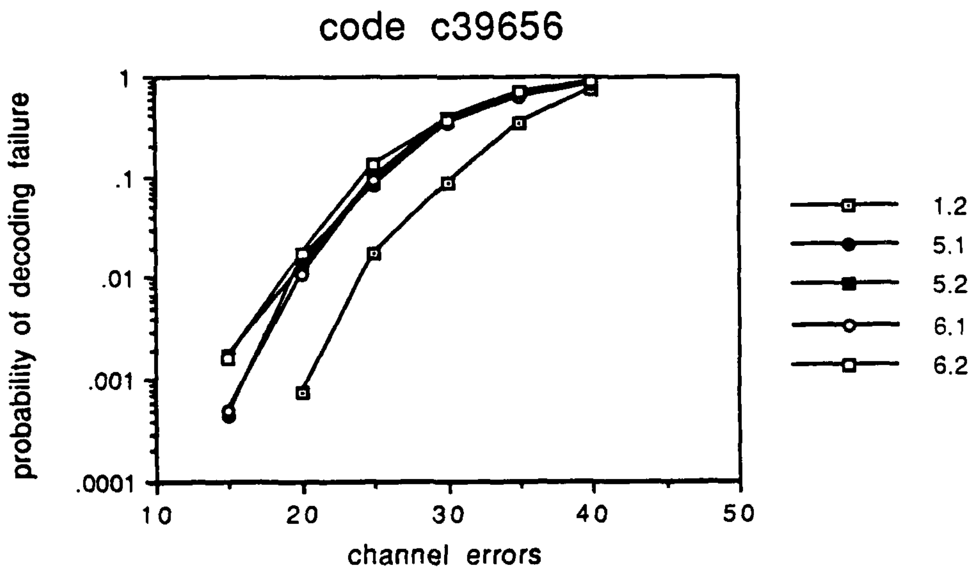
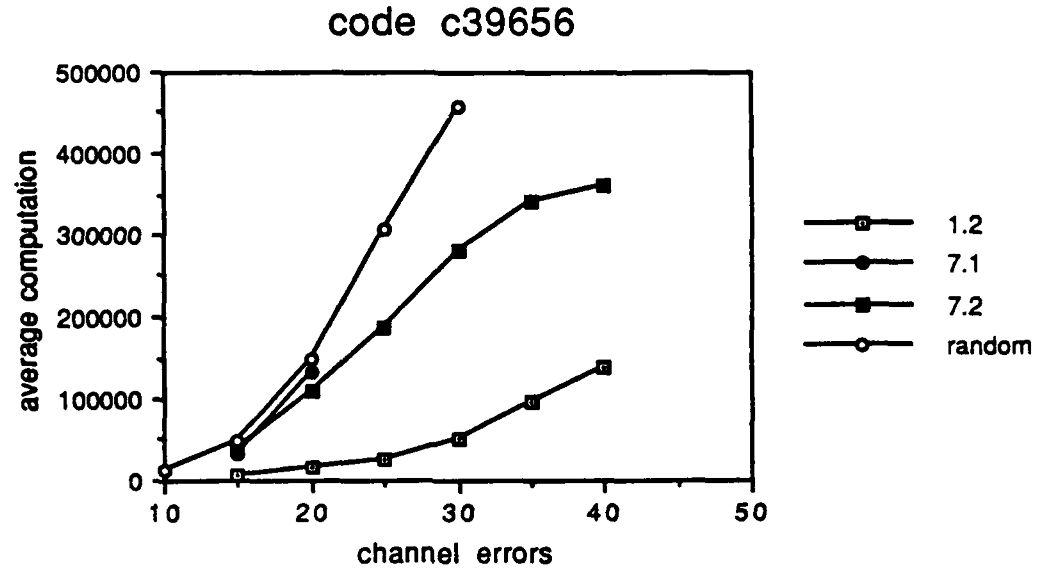


Figure 3.6. Simulation results for objective functions 1.2, 5.1, 5.2, 6.1, and 6.2 used with a (396,5,6) low-density code.



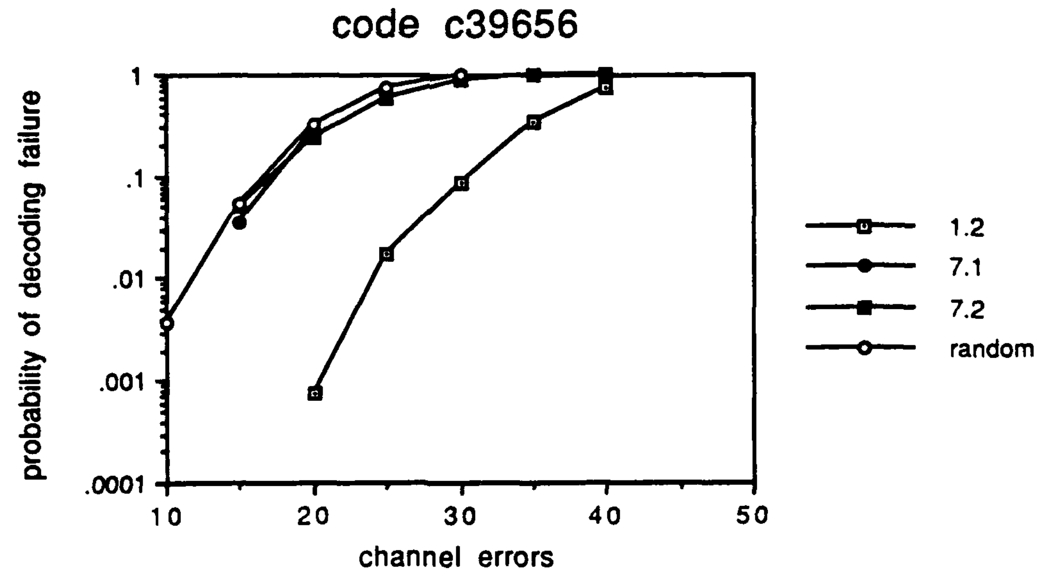
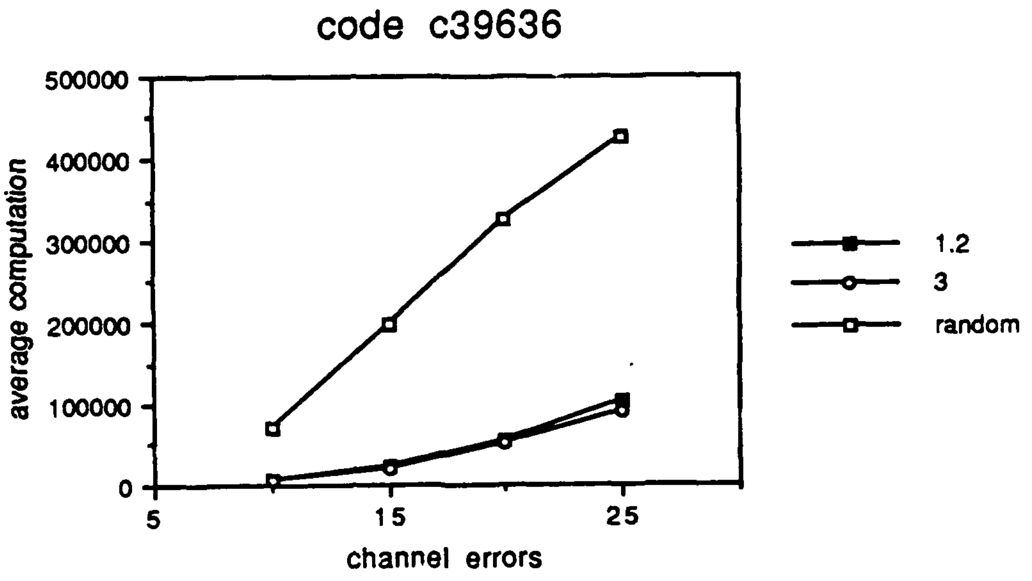


Figure 3.7. Simulation results for random ordering and objective functions 1.2, 7.1, and 7.2 used with a (396,5,6) low-density code.



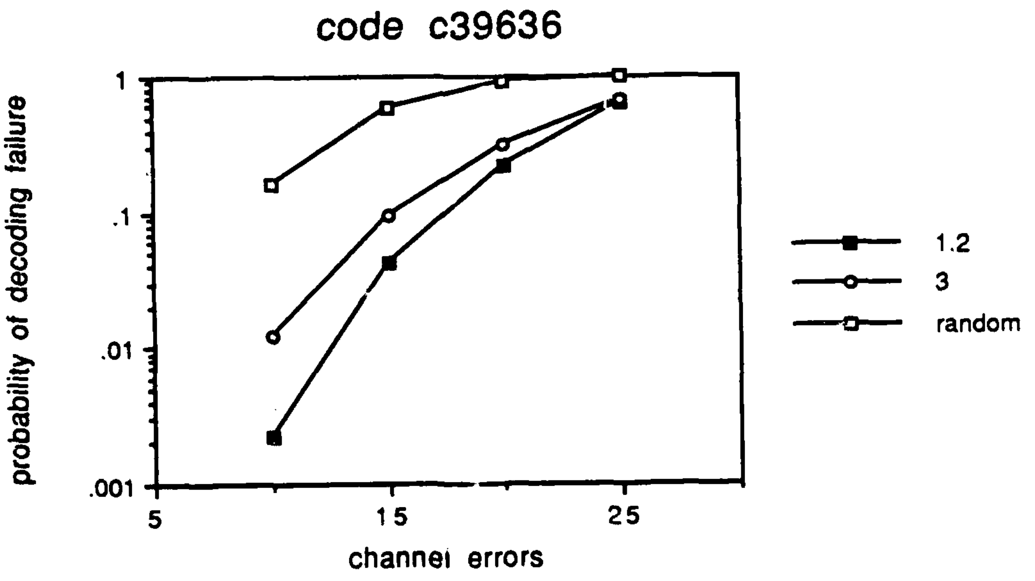
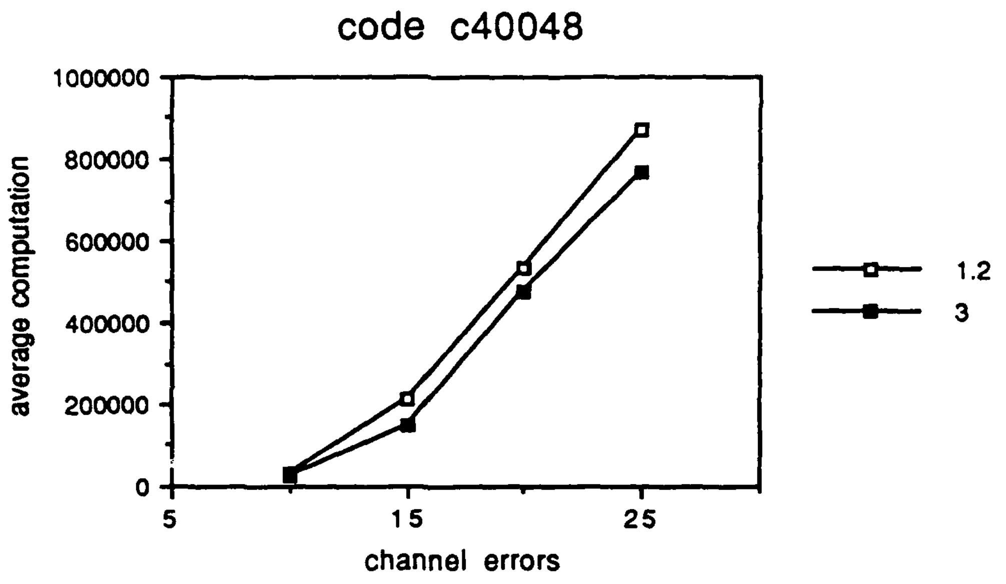


Figure 3.8. Simulation results for random ordering and objective functions 1.2 and 3 used with a (396,3,6) low-density code.



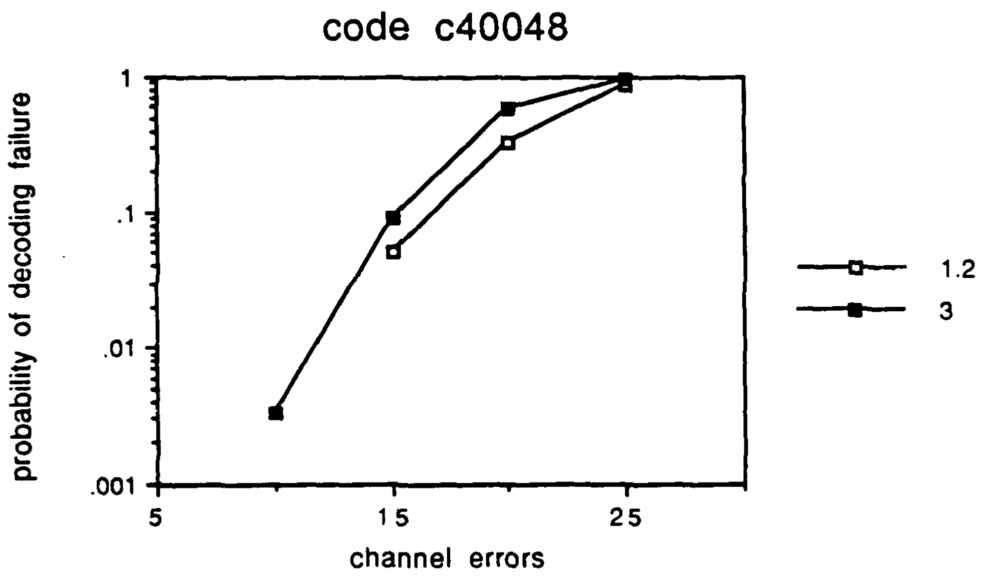
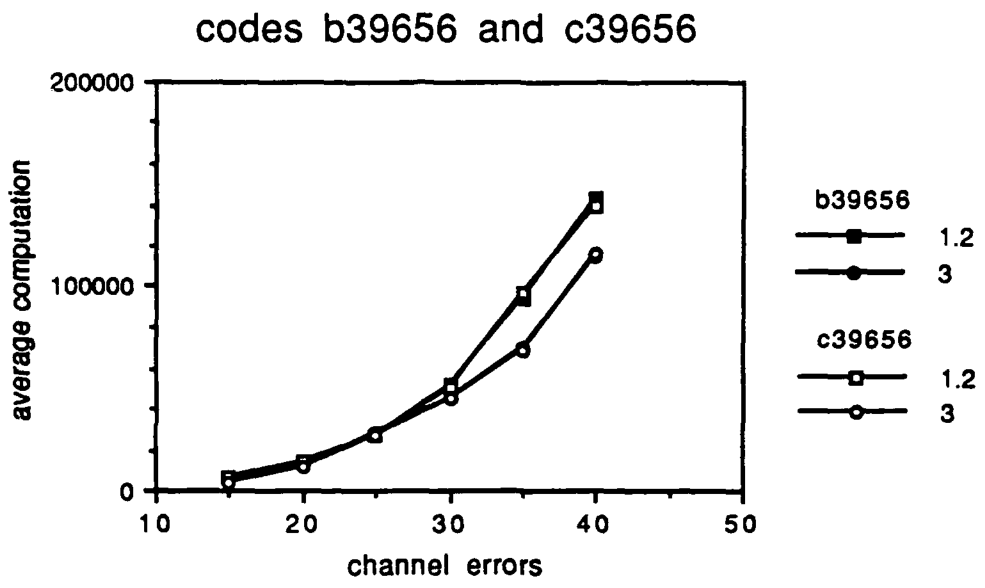


Figure 3.9. Simulation results for objective functions 1.2 and 3 used with a (400,4,8) low-density code. For objective function 1.2 with 10 channel errors, there were zero decoding failures out of 2477 trials.



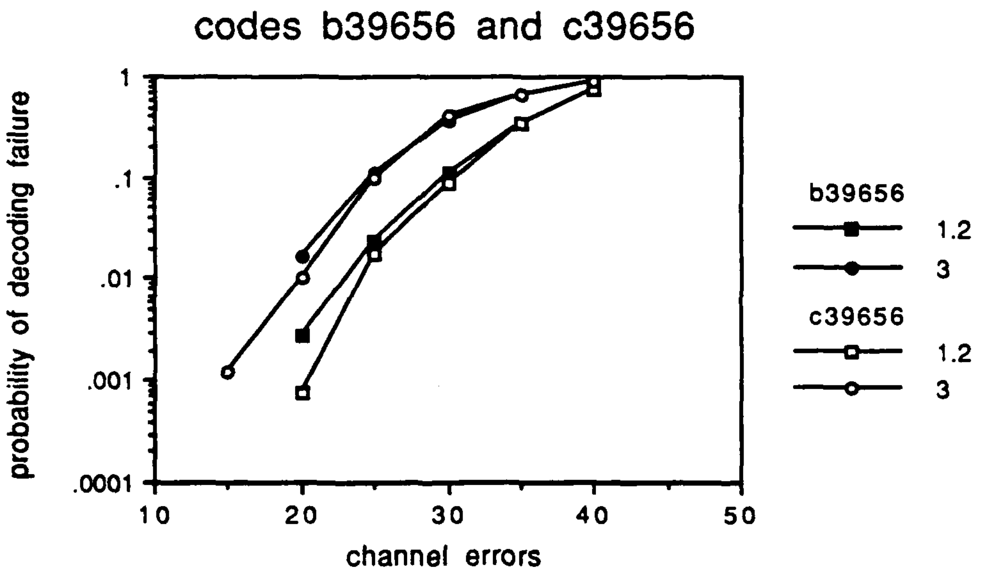


Figure 3.10. Simulation results for objective functions 1.2 and 3 used with two (396,5,6) low-density codes, b39656 and c39656.

used on the BEC. As a result, the computer recorded N_F, the number of declared decoding failures, as well as N_E, the number of decoding errors, and N_A, the number of aborted trials. In this case, a trial was aborted if the computation reached 10^6; this higher value was used because steps performed by the sequential decoder used here are simpler than those of the stack algorithm. The estimated probability of decoding failure is now given by $p_{DF} = (N_F + N_E + N_A)/T$, where T is the total number of trials.

The functions that minimized the average computation were 5.1 and 6.1, and the results for these two were nearly equal. Apart from the random ordering algorithm, 7.1 and 7.2 performed worst. The average computation for 7.2 was as much as 12 times as great as that of 6.1. However, all the objective functions performed better than random ordering, which required as much as 16 times the computation required by 6.1. With respect to p_{DF}, the probability of decoding failure, objective function 1.2 performed best, closely followed by 1.1. Random ordering was worst, followed by 7.1 and 7.2.

Recall that p_{DF} was defined to include decoding failures, decoding errors, and trials aborted due to too much computation. As with the BEC results, there was a surprising lack of decoding errors. A decoding error occurs when the decoder completes execution and produces an output, but the output is incorrect. Out of a total of 112,898 trials, only two resulted in decoding errors. Both errors occurred using objective function 1.2 with the (396,3,6) code and 25 channel errors. This was the weakest code used; also, p_{DF} was

0.634, hence this algorithm would probably not be used with this much channel noise anyway. Therefore, if these decoding algorithms are used on a channel with feedback and retransmission capabilities, one can achieve decoding error rates several orders of magnitude smaller than p_{DF}. However, the effective information rate would be lower than the code ıate, due to retransmissions.

The relative performance of random ordering and functions 1.2 and 3 observed with the (396,3,6) and (400,4,8) codes was the same as that observed with code c39656. This reinforces the hypothesis that an ordering algorithm that performs well for one value of j and k will also perform well for other values. The average computation for the (400,4,8) code was roughly eight times greater than that of the (396,3,6) code, while p_{DF} was greater for high channel noise and less for low channel noise. Note that both these codes have the same designed code rate, 1/2. Thus, it seems that increasing k will dramatically increase the average computation, even when the code rate is kept fixed.

With regard to b39656, the second (396,5,6) code, the results agreed very well with those obtained with c39656. This reinforces the claim that c39656 is a typical (396,5,6) low-density code. It should be stressed that b39656 and c39656 were chosen randomly, without regard for their performance.

From the simulation results, it appears that objective function 1.2 performed best overall. Its average computation was comparable to that of the minimum observed values, and its probability of decoding failure was lowest. This leads one to consider what characterizes a good objective function. It seems unlikely that V_i itself is an exceptionally good quantitative measure of digit reliability, even though it was used this way in function 1.2. Why should a digit involved in three violated parity checks be 1.5 times as unreliable as a digit involved in two? Instead, it seems that function 1.2 performed well because its measure of digit reliability was discrete. This allowed the ordering algorithm to consider a group of parity checks with relatively good digit reliabilities, and of these, choose the one with minimum $|N|$. With this approach, the digit reliability measure need not be extremely accurate, and the exact tradeoff between maximizing reliability and minimizing $|N|$ need not be specified. Thus, one would expect that discretizing any of the reliability measures used by the relatively good objective functions – namely, 3, 4, 5, and 6 – together with breaking ties by minimizing $|N|$, will result in a good ordering algorithm.

What conclusions can we draw concerning the feasibility of this approach at rates greater than R_0, the computational cutoff rate for sequential decoding? Note that this discussion applies only to the binary symmetric channel. First, we consider low-density codes with $j=5$ and $k=6$, which have designed code rate $R=1-j/k = 1/6$. It is well known that the formula for R_0, given in Section 1.1, applied to the BSC yields

$$R_0 = 1 - \log_2 [1+2\sqrt{p(1-p)}] \,, \tag{3.16}$$

where p is the channel error probability and R_0 is measured in bits per channel use. The value of p corresponding to $R_0 = 1/6$ is $p = 0.1882$. For blocklength $n = 396$, this leads to an expected number of errors per codeword given by $np = 74.5$. To decide how many errors per codeword can be handled by the best BSC algorithm, we used a different criterion than for the BEC. This was done because average computation on the BSC did not increase as abruptly as on the BEC. The criterion used here is that p_{DF} should be less than 0.1. Then, at most 30 errors per codeword can be handled by the best algorithm presented here. This is summarized in Table 3.1. Thus, it does not appear feasible to use these algorithms at rates greater than R_0 for rate 1/6 codes.

However, low-density codes with different values of j and k are more promising. Table 3.1 contains data for the (396,3,6) and (400,4,8) codes used here. Both have designed rate $R = 1/2$. As above, $p_{DF} < 0.1$ was used to determine feasibility. The results for these codes are very close to the maximum performance of sequential decoding. This is quite good, because the low-density codes were chosen at random, without regard for their performance, and the result for sequential decoding is a maximum for all tree codes. More study, including simulations using codes with longer blocklengths, would show more accurately whether these algorithms allow one to exceed R_0.

Table 3.1. Comparison of SDR algorithms and sequential decoding for the binary symmetric channel.

Code parameters			Rate	Achievable *np* with SDR (approximate)	Max. crossover prob. for sequential decoding	Max. *np* for sequential decoding
n	*j*	*k*	*R*			
396	5	6	1/6	30	0.1882	74.5
396	3	6	1/2	17	0.0449	17.8
400	4	8	1/2	17	0.0449	18.0

Since these algorithms perform better at rate 1/2 than rate 1/6, this leads one to question whether they perform even better at higher rates. More work is required to answer this question. It should be stressed that the results given here are only lower bounds for what is achievable with SDR algorithms. Other ordering algorithms, with possibly completely different structures, may perform better.

Some features that limit this approach, that are not encountered in traditional sequential decoding, include the following. As pointed out in Section 1.2, the codeword tree for a low-density code grows more rapidly near the beginning than near the end. This is a drawback for the same reasons discussed in Section 2.2. In addition, low-density codes may not have as good distance structure as arbitrary tree codes. Another important point is that the BSC is, in a sense, the worst possible channel for SDR algorithms. Each digit at the channel output, taken by itself, is equally unreliable – unlike, for example, the binary erasure channel and the additive white Gaussian noise channel. This makes it difficult to generate a good codeword tree ordering.

3.3 Backtracking Procedures and a New Sequential Decoding Algorithm

Without modification, the Fano sequential decoding algorithm (p. 269 of [5]) does not work well with low-density codes. The problem occurs when the decoder decides it is on the wrong path due to a wrong move earlier in the codeword tree. This happens when none of the branches at the current level of the tree are close to the corresponding received digits; each branch causes the path metric to fall below the threshold. In the Fano algorithm, the decoder moves back one level and tries to move laterally. If it cannot find an acceptable path, it moves back another level and again tries to move laterally. This continues until the decoder reaches the most recent level at which the threshold was raised, at which point it lowers the threshold and tries to move forward again.

This procedure works well for convolutional codes, because if the decoder makes a wrong move, the following levels of the codeword tree quickly become independent of the received data. Thus, when the decoder makes a wrong move, it will realize this within a few levels, and backtracking one level at a time will quickly find the problem. This situation does not hold for low-density codes. In a low-density code's codeword tree, it is possible to have the two subtrees following a correct move and an incorrect move be identical for many levels (for example, 50 or more levels in a 400 blocklength code).

This situation occurs because the labels of the branches at a given level in the codeword tree depend only on the values of the few digits in the level's o-set. Recall that each level in a low-density code's codeword tree corresponds to a parity check. The new digits at a level are those digits that are involved in the corresponding parity check, but not in any of the previous parity checks. The old digits at a level are the remaining digits involved in the level's parity check; they have already been assigned values earlier in the tree.

As in Section 1.2, the set of new digits at a level is called the level's n-set, and the set of old digits is called the o-set. The parity of a level's old digits determines what values can be assigned to a level's new digits.

When a wrong move occurs, the two subtrees originating from the wrong move and the correct move will be identical until one of the digits assigned incorrectly in the wrong move occurs in some level's o-set. It is true that the ordering algorithm favors parity checks with large o-sets, so that once a parity check containing a given digit is chosen, other parity checks containing that digit are more likely to be chosen. However, given an occurrence of a digit at a level in the codeword tree, there is no guarantee for when its next occurrence will be.

Instead of backtracking one level at a time, the decoding algorithm could jump back to one of the levels that directly influence the current level. These levels, called significant parents, are the places where the digits in the o-set of the current level were initially assigned. Since we consider only low-density codes for which no two parity check sets contain more than one digit in common, the number of preceding levels that directly influence the current level will equal the number of digits in the current level's o-set. However, since this means there will be typically more than one significant parent, it is not immediately apparent how to incorporate this into the Fano algorithm.

A sequential decoding algorithm that utilizes significant parents is presented in Section A.3. A description of the algorithm is given below, followed by a discussion of its features and a bound on its computational complexity.

3.3.1 The sequential decoding algorithm

This algorithm is similar to the Fano algorithm and unlike the stack algorithm (Section 7.2.7 of Clark and Cain [3]) in that it stores only a single path through the codeword tree. As in the Fano algorithm, the decoder calculates a metric value for each branch of the path it takes. The metric is given below.

If $n_i > 0$,

$$\Gamma_i(\mathbf{x}_i,\mathbf{y}_i) = \sum_{m=1}^{n_i} \left[\log_2 \frac{P(y_{i,m}, V_{i,m} | x_{i,m})}{P(y_{i,m}, V_{i,m})} - r_i \right]. \tag{3.17}$$

If $n_i = 0$,

$$\Gamma_i = \begin{cases} 0 & \text{if the parity check at level } i \text{ is satisfied,} \\ -\infty & \text{otherwise.} \end{cases} \tag{3.18}$$

where

$\Gamma_i(\mathbf{x}_i,\mathbf{y}_i)$ = the metric value of the branch at level i with label $\mathbf{x}_i = (x_{i,m})_{m=1}^{n_i}$, when the corresponding received digits are $\mathbf{y}_i = (y_{i,m})_{m=1}^{n_i}$

n_i = the number of new digits at level i in the codeword tree

$V_{i,m}$ = the error statistic associated with $y_{i,m}$

= the number of violated parity checks involving $y_{i,m}$

r_i = the local rate of the codeword tree at level i, defined below.

The probabilities are calculated by assuming that the $x_{i,m}$ are independent and $P(x_{i,m}=0) = P(x_{i,m}=1) = 1/2$. The relevant formulas are given in Section 3.1. This metric is the same as the Fano metric (given by L_n in Section VII of Fano [4]) except for the inclusion of $V_{i,m}$ and r_i, and the rule for determining Γ_i when $n_i = 0$.

The error statistic $V_{i,m}$ is discussed in Section 3.1. It is included as a form of side information to estimate the reliability of a received digit. In an (n, j, k) low-density code, there are only $j+1$ values that $V_{i,m}$ can assume. For each of these values, the log term above can have two values, depending on whether or not $y_{i,m}$ equals $x_{i,m}$. Thus, the log term will assume only $2(j+1)$ different values. They can be efficiently stored in a table and need not be recalculated.

The local rate, r_i, replaces the fixed bias term in the Fano metric. The bias term usually equals R, the code rate. The local rate is used because the codeword tree for a low-density code does not grow with constant rate, as discussed in Section 1.2. The local rate is given by

$$r_i = \frac{n_i - l_i - 1}{n_i}, \tag{3.19}$$

where n_i is defined as above, and l_i is the number of consecutive levels s preceding i with $n_s = 0$. In other words, $i - l_i - 1$ is the first level preceding i with $n_i > 0$.

This definition is used because of the heuristic justification for the Fano metric given in Section VII of [4]. It seems that this justification implicitly uses the fact

$$\sum_{t=1}^{d_i} B_t = \log S_i \ , \tag{3.20}$$

where d_i is the number of digits encountered in the tree up to and including level i, B_t is the bias term for digit t, and S_i is the size of the tree at level i. For a fixed rate tree code, $S_i = 2^{d_i R}$, thus $B_t = R$ satisfies (3.20). For a low-density code's tree, Equations (1.2) and (1.3) give

$$\log_2 S_i = \sum_{m=1}^{i} (n_i + \alpha_i - 1) \ , \tag{3.21}$$

where $\alpha_i = 1$ if parity check i is redundant, given the preceding parity checks, and $\alpha_i = 0$ otherwise. If one ignores α_i and divides the bias term contribution of a level equally among that level's digits, one obtains (3.19). For a randomly chosen code, redundant parity checks will be rare, thus ignoring α_i should not have a significant effect. It was found in simulations that using r_i as the bias term worked much better than using R.

The decoder can be in one of three states – Forward mode, Backtrack mode, or Reset mode. Initially, the decoder is in Forward mode. Starting from the root node of the codeword tree, the decoder moves forward one level at a time. At each level, the decoder chooses the first outgoing branch with nonnegative branch metric. Note that it considers the branch metric, not the cumulative metric. However, if the decoder reaches a level where all outgoing branches have negative metric values, it enters Backtrack mode.

The level where the decoder enters Backtrack mode is denoted by bt_level. The purpose of entering this mode is to find a new path in the tree that has nonnegative branch metric at bt_level. However, as discussed above, this cannot be done efficiently by backtracking one level at a time. Instead, the decoder makes a list of the significant parents of bt_level. It assumes it made a wrong move at one of the parent levels, and tries to correct this by making changes at each of these levels, one at a time. It starts by jumping back to the first parent on its list and choosing the first outgoing branch – unless the original path followed the first branch, in which case it chooses the second. Note that it makes this choice even if the resulting branch metric is negative.

The decoder then works its way back up to bt_level in the same way as in Forward mode, with the following exception. If it encounters a level where all outgoing branches have negative metric, it chooses the branch with greatest metric and continues forward, instead of starting a new backtrack procedure.

When the decoder reaches bt_level, it stores the value of the cumulative path metric.

This process is repeated for the other outgoing branches at the first parent level, and for all the parent levels, except that any branch choice which was on the original path is skipped. Also, the last branch setting tried for one parent must be reset before changing the next parent. The decoder then decides whether to accept any of these changes. It considers the change with greatest cumulative metric at bt_level. If this metric value is greater than the original, it accepts the change; otherwise, no change is accepted.

At this point, the decoder enters Reset mode. The last change is reset to its original value, and any accepted change is implemented. In addition, if a change is accepted, its value is recorded. This insures that the change is remembered if the decoder passes through the changed level again, due to another backtrack. Note this does not protect the level from further change if it occurs as the significant parent of a new bt_level.

If there is now an outgoing branch at bt_level with nonnegative metric, the decoder chooses the first such branch and reenters Forward mode. If all branch metrics are still negative, two actions are possible, depending on the value of n_i for i = bt_level. If $n_i = 0$, then the parity check at bt_level is still unsatisfied; the decoder declares a decoding failure and halts. If $n_i > 0$, the decoder chooses the outgoing branch with greatest metric and reenters Forward mode. This last event does not necessarily mean the decoder is on the wrong path; channel noise can and most probably will result in negative branch metrics even on the correct path.

The decoder continues in this fashion until it tries to move forward from the last level in the tree, at which point it is done. Its output is the valid codeword corresponding to its final path.

Various shortcuts were used in the implementation of this algorithm. For example, recall that one digit in a parent's n-set corresponds to a digit in the child's o-set. To limit computation, the decoder tried a branch setting at a parent only if the setting changed the value of the digit in its child's o-set. Otherwise, the branch metric at the child would not be raised, unless there was some other chain of parent-child relations leading from the parent to the child, an unlikely occurrence. For another shortcut, the decoder used only one pass to reset the last change at one parent and set the first change at the next.

3.3.2 Discussion

The sequential decoding algorithm described above is a fairly straightforward way to incorporate the idea of jumping back directly to a level's significant parents. However, several other possibilities were considered, and it seems appropriate to comment on some features of the present algorithm.

In what follows, a backtrack procedure is a decision by the decoder that it should change a branch setting at one of the significant parents of a level.

In the sequential decoding algorithm described above, there is never more than one active backtrack procedure at any given time. When the decoder enters Backtrack mode, it either accepts a change in its path or it decides that no change is necessary. In either case, the issue is resolved and the decoder will never again enter Backtrack mode using the same bt_level. The algorithm is designed this way because the situation would become quite complicated if multiple unresolved backtrack procedures were allowed. To see why, suppose that several unresolved backtrack procedures are active, and the decoder decides that it is on the wrong path. The decoder would have to decide which set of parents to continue changing or whether to start a new backtrack procedure. In addition, the different backtrack procedures could perturb each other. For example, a change made by one procedure could reassign a digit in the o-set of a parent in another procedure. This changes the set of possible branch settings available at the parent, which could cause a good change in the second procedure to look bad or cause the second procedure to skip over a good change.

Another feature of the decoding algorithm is that, unlike the Fano algorithm, the decision to start backtracking is based on branch metrics, not the cumulative path metric. As discussed before, the two subtrees leading from a correct move and an incorrect move may be the same for many levels. If the decoder makes a wrong move, but moves through the incorrect subtree along the path that matches the correct one, the path metric may accumulate a large positive increase. This could drown out negative branch metrics until it is too late to discover the initial wrong move.

Recall that in a backtrack procedure, all potential changes are tried before a final decision is made. This may seem wasteful; one could stop backtracking immediately after an acceptable choice is discovered. However, suppose that a level's position in the tree is very close to one of its parents. Then it is quite probable that a change at this parent would affect the metric value only at the parent level and the child level. If the branch metric at the parent level decreases, while the metric at the child level increases, the net effect could easily be positive even if the change is incorrect. Testing all possible changes will insure that the correct change is considered, if it exists. Ruling out changes in "close" parents would not work well either, because it was found that too many necessary changes were missed.

Another possible remedy for bad changes caused by close parents is to evaluate changes at a level beyond bt_level, denoted by stop_level. However, it is difficult to determine where stop_level should be located. One rule that was implemented was to make a list of levels with negative branch metrics beyond bt_level and choose stop_level to be the entry at a fixed position in the list. This tends to insure that there will be some coupling between the changes made to a close parent and the levels traversed by the decoder. However, it was found that wrong moves after bt_level introduced too much uncertainty into the path metric value at stop_level.

Finally, two other significant changes in the algorithm were considered. Recall that the branch metric value is $-\infty$ for an unsatisfied parity check at a level with $n_i = 0$. Instead, this could be a finite negative number, and one need not immediately declare a decoding failure if a backtrack procedure fails to remedy the situation. This would give the sequential decoder more chances to find any incorrect moves. However, this feature rarely decoded a vector that was not decodable by the original algorithm, and it usually required significantly more computation. Another possible change in the algorithm is to backtrack through more than one level of parents; that is, consider the parent of a parent, and so on. However, this would cause a huge increase in the number of changes to consider.

3.3.3 A computation bound

The sequential decoding algorithm described above has the nice property that the decoder always moves forward through the codeword tree, in a certain sense. More specifically, the decoder can enter Backtrack mode from any given level at most once. From this property, one can easily obtain a bound on the number of forward moves performed by the decoder. Here, a forward move means any time the decoder moves from one level to the next, regardless of whether the decoder is in Forward, Backtrack, or Reset mode. The bound is given below.

$$F < n^2(1-R)^2 k 2^{k-1} , \qquad (3.22)$$

where

$$\begin{aligned} F &= \text{the total number of forward moves} \\ &\quad \text{performed by the sequential decoder} \\ (n, j, k) &= \text{low-density code parameters} \\ R &= \text{the designed information rate of the code} \\ &= 1 - j/k . \end{aligned}$$

Proof: Suppose the decoder enters Backtrack mode with bt_level = i, where i is a given level in the codeword tree. Let f_i be the number of forward moves performed by the decoder before it reenters Forward mode. Then

$$f_i < i a_i 2^{k-1} + i , \qquad (3.23)$$

where a_i is the number of old digits at level i, that is, the number of digits assigned earlier in the tree. The first term is present because: (1) a_i is the number of parents of level i, (2) there are at most 2^{k-1} possible branch settings for each parent, and (3) the number of levels between a parent level and bt_level is strictly bounded by i. The second term bounds the number of forward moves performed in Reset mode.

A given level can be bt_level at most once, thus F_{BR}, the total number of forward moves performed during Backtrack or Reset modes, satisfies

$$F_{BR} \le \sum_{i=1}^{nj/k} f_i$$

$$< \sum_{i=1}^{nj/k} (ia_i 2^{k-1} + i)$$

$$\le \frac{1}{2} nj(nj/k+1)\, 2^{k-1} + \frac{1}{2}(nj/k)(nj/k+1)\,, \qquad (3.24)$$

since nj/k is the number of levels in the codeword tree, and $a_i \le k$ for all i. The number of forward moves performed during Forward mode, F_F, equals the number of levels in the codeword tree, or nj/k. Using $F = F_{BR} + F_F$ and $R = 1 - j/k$, we obtain the desired result. □

This bound is quite loose because it assumes that the decoder enters Backtrack mode from every level in the codeword tree. However, the bound is much smaller than the total number of nodes in the tree, which is on the order of $n2^{nR}$.

In practice, this decoding scheme would probably be used only if the noise level was low enough so that relatively few backtrack procedures would be necessary. The maximum noise level would depend on the code rate and the maximum tolerable amount of computation.

CHAPTER 4

CONCLUSIONS

4.1 Summary of Results

In this thesis, we presented several decoding algorithms, called sequential decoding with reordering (SDR) algorithms. These algorithms are modifications of standard sequential decoding; they have the general structure shown in Figure 1.2. These modifications were done in an attempt to operate at rates greater than R_0, the computational cutoff rate for sequential decoding.

The SDR algorithms have two parts - an ordering algorithm and a sequential decoder. The ordering algorithm chooses an ordering of the parity checks that define the code being used. The sequential decoder uses this ordering to generate a codeword tree and searches this tree to obtain the decoder output. The SDR algorithms are used with low-density codes, a class of block codes. In this respect they differ from standard sequential decoding, which can be used with any tree code. Low-density codes are used because one can reorder their associated codeword trees, and the resulting trees have bounded growth rate.

We presented SDR algorithms for two channels - the binary erasure channel (BEC) and the binary symmetric channel (BSC). For the BEC, an upper bound was derived for the number of nodes visited by the sequential decoder, as a function of the parity check ordering and the channel erasure pattern. This bound motivated the construction of an ordering algorithm, called the minimum new erasure (MNE) algorithm.

The MNE algorithm was used in simulations with the stack algorithm, a standard sequential decoding algorithm. The MNE algorithm performed much better than random ordering. However, when used with a (396,5,6) low-density code, with designed rate $R = 1/6$, and a (396,4,6) low-density code, with $R = 1/3$, the MNE algorithm did not exceed the R_0 bound, which represents the best that standard sequential decoding can achieve. The performance was better for a (396,3,6) code, with $R = 1/2$. In this case, the algorithm roughly matched the R_0 bound. Simulations with longer blocklengths would indicate more clearly whether the R_0 bound is exceeded.

Also on the BEC, simulations verified the hypothesis that for a fixed designed code rate, the erasure correcting ability of the MNE algorithm increases linearly with blocklength. This behavior is characteristic of standard sequential decoding, but is not common among existing practical decoding schemes for block codes. In addition, in the feasible erasure region for the MNE algorithm, the average computation of the sequential decoder portion increased linearly with blocklength.

For the BSC, several ordering algorithms were presented, as well as a new sequential decoder. All the ordering algorithms have structures similar to that of the MNE algorithm; they differ in their choice of objective function. A list of various objective functions is shown in Figure 3.2. Through simulation, it was found that function 1.2 performed best. It performed significantly better than random ordering. Using a (396,5,6), $R = 1/6$ low-density code, the best BSC algorithm did not exceed the R_0 bound. However, as with the BEC, performance improved at higher code rates. For both a (396,3,6) and a (400,4,8), each with $R = 1/2$, the R_0 bound was roughly matched. Again, simulations with longer blocklengths would provide more information.

For both the BEC and BSC algorithms, decoding errors were much less frequent than decoding failures. This feature could be used to advantage on a channel with feedback and retransmission capabilities.

The BSC algorithms were used with a new sequential decoding algorithm, described in Section 3.3. In a low-density code's codeword tree, the two subtrees originating from a correct move and an incorrect move may appear the same for many levels, apart from the different assignments made at the level where they originally diverge. As a result, backtracking one level at a time, as in the Fano algorithm, can be very inefficient. For a low-density code, the labels at a given level in the codeword tree depend on the assignments at a small group of preceding levels, called the "significant parents." The new sequential decoding algorithm is able to backtrack directly to a level's significant parents. As a result, the algorithm may be applicable to other tree search problems where, instead of a single level, it is desirable to consider several levels as possible backtrack destinations.

The sequential decoder uses a modified version of the Fano metric. It differs from the standard Fano metric in that it contains a variable local rate term and also depends on the digit reliabilities computed by the ordering algorithm. Finally, two other features distinguish this algorithm from both the Fano and stack algorithms. It can declare a decoding failure, and its worst-case computation is $O(n^2)$, where n is the blocklength.

Reasons why SDR algorithms do not automatically outperform standard sequential decoding include the following. The SDR algorithms are constrained to use only low-density codes, while standard sequential decoding can use any tree code. As discussed in Section 1.2, low-density code codeword trees do not grow at a constant rate; they tend to grow quickly near the beginning and slowly near the end. This is a drawback for reasons discussed in Section 2.2. In addition, the features that required the use of a new sequential decoding algorithm make sequential decoding difficult for low-density codes. In addition, low-density codes probably do not have as good distance properties as arbitrary tree codes. Finally, the BSC is a particularly bad channel for SDR algorithms. Each digit at the channel output, taken by itself, is equally reliable. As a result, the ordering algorithm does not have much information to work with.

In conclusion, the performance of the SDR algorithms with rate 1/2 codes is encouraging. The R_0 bound for standard sequential decoding is an upper bound that applies to all tree codes, while the SDR results were obtained using randomly chosen codes. In addition, since the relative performance improved at higher code rates, SDR algorithms may outperform standard sequential decoding at rates greater than 1/2. Note that if a low-density code with rate greater than 1/2 is used, the code parameter j should be greater than 2, as discussed in Section 2.2. Finally, the results presented in this work are lower bounds to what can be achieved with SDR algorithms. Other ordering algorithms may perform better.

4.2 Directions for Further Research

Most of the results presented in this work were obtained through simulations. More could be said about asymptotic behavior if one obtained useful analytic bounds on the expected computation and the probability of decoding failure. It may be easier to determine average values for an ensemble of low-density codes, instead of for a specific code.

Some more specific suggestions include the following. As discussed in Section 4.1, it appears promising that SDR algorithms may exceed the R_0 bound when using code rates greater than 1/2, and it would be worthwhile to test this hypothesis.

All of the ordering algorithms presented here, for both the BEC and BSC, choose parity checks without considering their effect on later levels in the codeword tree. As an alternative, one could consider c parity checks at a time, where c is a constant. For example, the MNE algorithm chooses parity check i to minimize N_i. (See Section 2.1.) Using this approach, the modified MNE algorithm would determine the ordered set of c parity checks that minimize $N_i + \cdots + N_{i+c-1}$. Only the first parity check in the set is chosen, so th-+ each choice is made with the same amount of foresight. This approach would require more computation than the original MNE algorithm, but if the performance improved, it would allow one to trade off computation in the ordering algorithm for computation in the sequential decoder.

For the BEC, one could use the sequential decoder described in Section 3.3 instead of the stack algorithm. For the BSC, one could perform more than one iteration of Gallager's decoding algorithm to generate digit reliabilities. Again, this could be used as a computation tradeoff between the ordering algorithm and the sequential decoder. Finally, a promising SDR algorithm for arbitrary binary input memoryless channels is presented in Section 4.3.

4.3 An SDR Algorithm for Arbitrary Binary Input Memoryless Channels

In this section, we outline an SDR algorithm that can be used with any binary input memoryless channel. This class of channels includes the BEC and BSC, discussed in Chapters 2 and 3, respectively, as well as channels with side information and channels with real-valued outputs, such as the additive white Gaussian noise (AWGN) channel and the Rayleigh fading channel.

Recall that SDR algorithms consist of two parts – an ordering algorithm and a sequential decoder. The ordering algorithm used here, called the entropy algorithm, is described below. The presentation loosely follows the C programming language. First, we make the following definitions.

(n, j, k) : Parameters of the low-density code being used.

$\mathbf{x} = (x_i)_{i=1}^n$: The transmitted codeword.

$\mathbf{y} = (y_i)_{i=1}^n$: The channel output.

$m = nj/k$: The number of parity checks used to define the code.

$\{C_i,\ 1 \le i \le m\}$: The parity check sets that define the code, in arbitrary order. Each $C_i \subseteq \{1,2,\dots,n\}$. The parity equation corresponding to C_i is $\sum_{a \in C_i} x_a = 0 \pmod 2$.

$\{C_i',\ 1 \le i \le m\}$: An ordered list of the parity check sets.

f : A function on the channel output space, given by

$$f(y_a) = h(\mathrm{P}(x_a = 0 \mid y_a)), \tag{4.1}$$

where h is defined below and P is the probability distribution generated by the channel model and the a priori input distribution. For a channel with side information, the extra information would be included together with y_a in the formula for f.

h : The entropy function, given by

$$h(x) = -x\log x - (1-x)\log(1-x). \tag{4.2}$$

Input $n,\ j,\ k,\ \{C_i,\ 1 \le i \le nj/k\}$, $\mathbf{y}$

Output $\{C_i',\ 1 \le i \le nj/k\}$

The Entropy Algorithm

```
{
    $m = nj/k$;
    $A = \{1,2,\ldots,n\}$;
    $B = \{1,2,\ldots,m\}$;

    for ($i = 1$; $i <= m$; $i = i+1$) {

        $l = \arg\min_{b \in B} [ \sum_{a \in C_b \cap A} f(y_a) ]$;
        $C_i' = C_l$;
        $A = A \setminus C_l$;
        $B = B \setminus \{l\}$;

    }
}
```

This algorithm has the same structure as the ordering algorithms presented in Chapters 2 and 3.

The entropy algorithm can be combined with any sequential decoding algorithm to obtain a complete SDR algorithm. However, we recommend the sequential decoding algorithm described in Section 3.3, which is designed specifically for low-density codes. Pseudocode for this algorithm is included in Section A.3.

The entropy algorithm looks promising because it includes as special cases the MNE algorithm (Section 2.1) for the BEC, and objective function 1.3 (Section 3.1) for the BSC, both of which were found to work well. Also, recall that one desirable objective of an ordering algorithm is to put parity checks containing reliable new digits near the front of the codeword tree. The entropy algorithm will tend to satisfy this objective, because the entropy function is a natural additive measure of unpredictability. Finally, SDR algorithms may work better on channels with real-valued outputs, such as the AWGN, than on the BEC and BSC, because the output can attain a greater number of reliability levels. This gives an ordering algorithm more information to work with. In a sense, the V_i statistic, defined in Section 3.1, is an artificial way of generating more reliability levels for the output of the BSC.

The decoding algorithm described here can be used as a soft decision decoder, because the form of the channel output is not constrained. Soft decision decoders usually perform better than hard decision decoders, because less information is thrown away by the demodulator. Recall that all SDR algorithms, including the one described in this section, are designed for use with low-density codes, a class of block codes. Block code decoding algorithms that handle soft decisions are uncommon.

At this point, we present some potential improvements in the entropy algorithm. To begin with, it may be advisable to quantize the function f. This can significantly reduce the difficulty of evaluating f. Next, define $E(b) = \sum_{a \in C_b \cap A} f(y_a)$. Whether or not we quantize f, there are good reasons for quantizing E. First, this can lead to better orderings, as discussed in Section 3.1. Second, we can reduce the computational complexity of the entropy algorithm from $O(n^2)$ to $O(Vn)$, where V is the number of distinct values E can assume, and j and k are fixed. This reduction is accomplished the same way as for the BSC ordering algorithms and is outlined in Section 3.1.

In conclusion, the decoding algorithm presented here is designed for low-density codes and can be used with any binary input memoryless channel. In particular, it can be used on channels with side information and channels with real-valued outputs and as a soft decision decoder.

APPENDIX
LIST OF ALGORITHMS

A.1 An Algorithm to Generate Low-Density Codes

A.1.1 Preliminary remarks

The algorithms presented in the Appendix are described in psuedocode, using a syntax loosely based on the C programming language. Some features that may be unfamiliar to those unacquainted with C include the following. The operator "!=" means "is not equal to," and "==" means "is equal to." Arrays begin at index zero, unlike Fortran. The presentation here differs from C in several ways. For example, the words "or" and "and" are used instead of "||" and "&&." Also, most argument lists for functions are not included, and some operations are described in words instead of computer code. Function names are printed in boldface, and variables in italics.

The algorithm below attempts to generate the parity equations that define a low-density code with parameters (n, j, k). The quantities n, j, and k must be positive integers, with $j < k$ and n a multiple of k. The parity equations are chosen to satisfy the additional condition that no two of them involve the same group of two or more digits. It may not be possible to generate such a low-density code for given values of n, j, and k. For this reason, the algorithm will halt if *calls* > *call_max*. (A description of all variables is included below.)

The algorithm generates the parity equations in blocks of size n/k, as discussed in Section 1.3. Each digit appears in exactly one parity equation in each block. Parity equations are associated with rows in the parity matrix. The algorithm tries to fill each row using a random selection from the available digits. A digit is not available if it appears earlier in the current block, or if choosing it would cause two rows to overlap in more than one digit. If there are no more available digits, the algorithm clears the row and starts over. It tries to fill a row at most *r_max* times, at which point it starts over at the beginning of the current block. It tries to fill a block at most *b_max* times, at which point everything is cleared and the algorithm starts over from scratch. This continues until either a code is successfully generated ($status = 0$), or $calls > call\_max$ ($status = 1$). The codes used in this work were generated using $b\_max = r\_max = 25$.

The algorithm makes use of an external function **rand()**. It is assumed that invoking **rand**(*seed*), where *seed* is a non-zero floating point value, will initialize a psuedorandom number generator. Subsequent calls of **rand**(0.) will return a floating point value uniformly distributed over the interval [0,1). This corresponds to the function **rand()** provided with UNIX Fortran.

A.1.2 Description of variables

b_max	The maximum allowed value of *block_tries*.
block	The number of the current block being filled. Blocks start at zero.

block_failure	A flag that indicates *block_tries* > *b_max*.
block_filled	A flag that indicates the current block has been successfully filled.
block_tries	The number of times the algorithm has tried to fill the current block.
call_max	The maximum allowed value of *calls*.
calls	The number of times **rand**() is called, not counting the initial call of **rand**(*seed*).
flag[*i*]	A flag that indicates digit i has been included in a row in the current block.
*flag*2[*i*]	An array of flags that indicate forbidden digits for the current row being filled.
i,	Loop variables.
index	The number of digits included so far in the current row.
j	A low-density code parameter. The code being generated has parameters (n, j, k).
k	Another low-density code parameter.
n	The blocklength of the code.
new_digit	The new digit added to the current row.
open	The number of available digits that may be added to the current row.
other[*i*][*s*][*i*2]	An array used to insure that no two rows have more than one common digit. *other*[*i*][*s*][*i*2] is the index of the i2-nd digit in the row in block i containing digit s.
par[*i*][*i*2]	This array stores the parity checks that define the code being generated. *par*[*l*][*i*] is the index of the i-th digit involved in the l-th parity check. Digits are numbered from 0 to $n-1$. The first parity check has index one, not zero, in order to be compatible with the decoding algorithm of Section A.3.
rand()	An external function that generates a random number uniformly distributed over [0,1). It is described more fully in the **Preliminary remarks.**
row	The number of the current row being filled. Rows start at zero.
row_failure	A flag that indicates *row_tries* > *r_max*.
row_filled	A flag that indicates the current row has been successfully filled.

row_tries	The number of times the algorithm has tried to fill the current row.
s, *s*2	Integer temporary variables.
seed	Initializes the random number generator.
status	The output status of the algorithm. 0 = Code successfully generated, 1 = Algorithm stopped because *calls* > *call_max*.
stop	A flag that indicates *calls* > *call_max*.

A.1.3 Storage requirements

All variables are integer valued, with the exception of *seed*, which is floating point valued. The function **rand**() returns floating point values. The arrays have the following sizes:

$flag[n]$, $flag2[n]$, $other[j][n][k]$, $par[nj/k+1][k]$

A.1.4 The algorithm

Input	*n*, *j*, *k*, *seed*, *b_max*, *r_max*, *call_max*
Output	*par*, *status*, *calls*

Main program

```
{       /* Begin. */

stop = 0;
block = 0;

/* Initialize random number generator. */
rand(seed);

while (block < j and stop == 0) {

        Construct a block;
        if (block_filled == 1)
                block = block+1;
        else
                block = 0;

}
if (block == j)
        status = 0;
else
        status = 1;

}       /* End. */
```

Construct a block

```
{       /* Begin. */

block_tries = 0;
block_filled = 0;

while (block_filled == 0 and block_tries < b_max and stop == 0) {

        block_tries = block_tries+1;

        block_failure = 0;
        row = block*n/k;
        for (i = 0; i < n; i = i+1)
                flag[i] = 0;

        while (row < (block+1)*n/k and block_failure == 0 and stop == 0) {

                Construct a row;
                if (row_filled == 1)
                        row = row+1;
                else
                        block_failure = 1;

        }

        if (row == (block+1)*n/k)
                block_filled = 1;

}
return;

}       /* End. */
```

Construct a row

```
{       /* Begin. */

row_tries = 0;
row_filled = 0;

while (row_filled == 0 and row_tries < r_max and stop == 0) {

        row_tries = row_tries+1;

        index = 0;
        row_failure = 0;
        for (i = 0; i < n; i = i+1)
                flag2[i] = 0;

        while (index < k and row_failure == 0 and stop == 0) {

                open = 0;
                for (i = 0, i < n; i = i+1)
                        if (flag[i] == 0 and flag2[i] == 0)
                                open = open+1;

                if (open > 0) {

                        calls = calls+1;
                        if (calls >= call_max)
                                stop = 1;

                        /* Pick next digit. */
                        /* The value of s is a random integer, uniformly
                        distributed between 1 and open. */

                        s = integer part of (rand(0)*open+1);
                        new_digit = -1;
                        s2 = 0;
                        while (s2 < s) {
                                new_digit = new_digit+1;
                                if (flag[new_digit] == 0 and flag2[new_digit] == 0)
                                        s2 = s2+1;
                        }
                        par[row+1][index] = new_digit;
                        /* row+1 is used because par starts at one, not zero. */
                        /* Update flag2. */
                        for (i = 0; i < block-1; i = i+1)
                                for (i2 = 0; i2 < k; i2 = i2+1) {
                                        s = other[i][new_digit][i2];
                                        flag2[s] = 1;
                                }
                        flag2[new_digit] = 1;

                        index = index+1;

                }
                else
                        row_failure = 1;

        }

        if (index == k) {

                /* This row is complete. */
                row_filled = 1;

                /* Update flag and other. */
                for (i = 0; i < k; i = i+1) {
                        s = par[row+1][i];
                        flag[s] = 1;
                        for (i2 = 0; i2 < k; i2 = i2+1)
                                other[block][s][i2] = par[row+1][i2];
                }

        }

}
return;

}       /* End. */
```

A.2 An $O(n)$ Implementation of the MNE Algorithm

A.2.1 Preliminary remarks

The syntax used to present this algorithm is discussed in Section A.1, and the MNE algorithm is described in Section 2.1. Briefly, the algorithm takes as input the channel output vector and the set of parity checks that define the low-density code being used. Initially, the parity checks are in arbitrary order. The algorithm's purpose is to generate a new ordering, which is later used by a separate sequential decoding algorithm to generate a codeword tree.

Parity checks are referred to by their position in the original ordering. Each parity check is associated with a quantity called its *e_value*, defined to be the number of new erasures involved in the parity check. The algorithm generates the new ordering by choosing parity checks one at a time. At each step, it chooses the remaining parity check with smallest *e_value*.

To minimize computation, the parity checks are grouped into "bins." Each bin contains the labels of all the parity checks with a given *e_value*. To start, the algorithm places each parity check into the proper bin. Subsequently, the new ordering is generated by choosing parity checks from the first nonempty bin with smallest *e_value*. However, when a parity check is chosen, this may change the *e_value* of some remaining parity checks. This happens because the definition of "new" erasures depends on which parity checks have already been chosen. As a result, the algorithm updates the bins after each choice.

Each bin is organized as a two-way linked list, with pointers stored in the arrays *forward* and *backward*. Note *forward* and *backward* store integers, not memory address locations; see the Description of Variables below. The array *top* stores the location of the most recent addition to each bin. If a bin is not empty, new elements are placed at location *top*+1. This is done even if there are earlier empty locations, to save search time. No bin is required to use more than nj/k locations. This happens because nj/k is the total number of parity checks, and no parity check can enter a bin more than once, since a parity check's *e_value* can never increase.

A.2.2 Description of variables

backward[i][$i2$]	The first nonempty location preceding location $i2$ in bin i. If $i2$ is the first nonempty location, *backward*[i][$i2$] $= -1$.
bin[i][$i2$]	This array holds the bins. The bin number is i, and the location within the bin is $i2$.
e_value[i]	The number of new erasures contained in parity check i.
exit	A flag used to control program flow.
flag[i]	An array of flags that indicate which digits are contained in the parity checks chosen so far in the new ordering.
forward[i][$i2$]	The next nonempty location after location $i2$ in bin i. If $i2$ is the last nonempty location, *forward*[i][$i2$] $= -1$.
i, $i2$, $i3$	Loop variables.
j	A low-density code parameter. The code being used has parameters (n, j, k).

k	A low-density code parameter. See j.
label	Temporarily stores the contents of a bin location.
level_max	The number of parity checks used to define the code. $level\_max = nj/k$.
location$[i]$	The bin location of parity check i. Label i is stored at $bin[e\_value[i]][location[i]]$.
n	The blocklength of the code.
new_top	The new value of *top* for the current bin.
next	The label of the next parity check in the new ordering.
old_top	The old value of *top* for the current bin.
par$[i][i2]$	This array stores a list of the parity checks that define the code, in the new ordering as determined by the algorithm. See *unpar*.
pc_label$[i][i2]$	The label of the $i2$-nd parity check containing digit i, where $i2$ starts at zero.
rword$[i]$	This array stores the received word, obtained at the output of the communication channel. When $rword[i] = 2$, this indicates an erasure at digit i.
s, $s2$	Integer temporary variables.
sum$[i]$	An array of counters, used to generate *pc_label*.
top$[i]$	The location of the last element added to bin i. $top[i] = -1$ if bin i is empty.
unpar$[i][i2]$	This array stores a list of the parity checks that define the code, in the initial arbitrary ordering. $unpar[i][i2]$ is the index of the $i2$-nd digit involved in the i-th parity check. Parity checks are numbered starting at one, not zero, to agree with the presentation in Section A.3. Digits are numbered from zero to $n-1$.

A.2.3 Storage requirements

All variables are integer valued. The arrays have the following sizes, where $lm = level\_max+1 = nj/k+1$.

$backward[k+1][lm]$, $bin[k+1][lm]$, $e\_value[lm]$, $flag[n]$, $forward[k+1][lm]$, $location[lm]$, $par[lm][k]$, $pc\_label[n][j]$, $rword[n]$, $sum[n]$, $top[k+1]$, $unpar[lm][k]$

A.2.4 The algorithm

Input *n*, *j*, *k*, *unpar*, *rword*

Output *par*

Main Program

```
{       /* Begin */

Initialization;

for (i = 1; i <= level_max; i=i+1) {

        /* Look for first nonempty bin. */
        i2 = -1;
        exit = 0;
        while (exit = 0) {
                i2=i2+1;
                if (top[i2] != -1)
                        exit = 1;
        }

        /* Choose next parity check. */
        next = Pop(i2);
        for (i2 = 0; i2 < k; i2=i2+1)
                par[i][i2] = unpar[next][i2];

        /* Update bins. */
        for (i2 = 0; i2 < k; i2=i2+1) {

                s = unpar[next][i2];
                if (flag[s] == 0) {

                        flag[s] = 1;
                        if (rword[s] == 2) {

                                for (i3 = 0; i3 < j; i3=i3+1) {

                                        s2 = pc_label[s][i3];
                                        if (s2 != next) {
```

```
                                    Remove(e_value[s2], location[s2]);
                                    e_value[s2] = e_value[s2]-1;
                                    location[s2] = Push(s2, e_value[s2]);
                              }
                        }
                  }
            }
      }
}
}     /* End */
```

Initialization

```
{     /* Begin */

s = n*j;
level_max = s/k;
/* Computing level_max in two steps insures that multiplication is
performed before division, so truncation errors will not result from
integer division. */

/* Initialize flag. */
for (i = 0; i < n; i = i+1)
      flag[i] = 0;

/* Generate pc_label. */
for (i = 0; i < n; i = i+1)
      sum[i] = -1;

for (i = 1; i <= level_max; i = i+1) {
      for (i2 = 0; i2 < k; i2 = i2+1) {
            s = unpar[i][i2];
            sum[s] = sum[s]+1;
            pc_label[s][sum[s]] = i;
      }
}

/* Initialize top. */
for (i = 0; i <= j; i = i+1)
      top[i] = -1;
/* Initialize bins. */
for (i = 1; i <= level_max; i = i+1) {
        s = 0;
        for (i2 = 0; i2 < k; i2= i2+1)
                if (rword[unpar[i][i2]] == 2)
                        s = s+1;
        e_value[i] = s;
        location[i] = Push(s,i);
}

}       /* End */
```

Push(s,i)

```
/* Place parity check s into bin i, and return its location. */

{       /* Begin */

if (top[i] == -1) {

        /* This bin is empty. */
        bin[i][0] = s;
        top[i] = 0;
        forward[i][0] = -1;
        backward[i][0] = -1;
        return(0);

}
else {

        /* This bin is not empty. */
        old_top = top[i];
        new_top = old_top+1;
        bin[i][new_top] = s;
        top[i] = new_top;
        forward[i][old_top] = new_top;
        forward[i][new_top] = -1;
        backward[i][new_top] = old_top;
        return(new_top);

}

}       /* End */
```

Pop($i2$)

```
/* Remove the top element from bin $i2$, and return its contents. */

{     /* Begin */

if ($top[i2] == -1$)
      return($-1$);

else {

      $label = bin[i2][top[i2]]$;
      $new\_top = backward[i2][top[i2]]$;
      $top[i2] = new\_top$;
      if ($new\_top$ != $-1$)
            $forward[i2][new\_top] = -1$;
      return($label$);

}

}     /* End */
```

Remove($t, t2$)

```
/* Remove the element at location $t2$ in bin $t$. */

{     /* Begin */

if ($top[t] == t2$)
      $top[t] = backward[t][t2]$;

if ($forward[t][t2]$ != $-1$)
      $backward[t][forward[t][t2]] = backward[t][t2]$;

if ($backward[t][t2]$ != $-1$)
      $forward[t][backward[t][t2]] = forward[t][t2]$;

}     /* End */
```

A.3 A New Sequential Decoding Algorithm for Low-Density Codes

A.3.1 Preliminary remarks

Section 3.3 discusses the behavior of this algorithm, and Section A.1 contains some notes on the syntax of the description given below. The algorithm is designed for the Binary Symmetric Channel, but it can be modified to handle other binary input channels by changing the formula for the metric, m. With respect to the simulation results of Section 3.2, forward and lateral moves are defined as follows. A forward or lateral move occurs whenever the "while" loop in function **Kernel**, presented below, is executed. When **Kernel** is called, the first execution of the while loop is a forward move, and the other executions are lateral moves. These moves are functionally similar to forward and lateral moves performed by the Fano sequential decoding algorithm [5]. In what follows, the word "level" refers to a level in the codeword tree. The terms n-set and o-set are defined in Section 1.2.

A.3.2 Description of variables

$a[i][i2]$	This array contains a list of all binary vectors of length $k-1$. $a[i][i2]$ is digit $i2$ of vector i.
active	In Backtrack mode, the index of the active parent; that is, *parent*[*active*] is the parent being changed. Before any parent is chosen, *active* = -1.
b	The current branch number, used in **Kernel**.
b_flag[i][$i2$]	A flag set to 1 if *branch*[*parent*[i]] is set to $i2$ and the resulting change is followed through up to *bt_level*.

best, *best*2	Stores the best change discovered in Backtrack mode. This change is obtained by setting *branch*[*parent*[*best*]] = *best*2.
beta	An intermediate quantity used to calculate metric values.
branch[*l*]	The branch number at level *l* chosen by the decoder.
branch_best	The branch number with metric value *met_best*.
branch_max[*l*]	The maximum branch number at level *l*. Branches are numbered starting at zero, so *branch_max*[*l*] is one less than the number of outgoing branches leaving a node at level *l*—1.
bt_level	The level in the codeword tree from which Backtrack mode is entered.
cross_ref[*i*]	The level where digit *i* is assigned; in other words, the level where digit *i* occurs in the n-set.
d[*l*]	The size of the n-set at level *l*.
digit[*i*]	In Backtrack mode, *digit*[*i*] stores the value of the digit in the o-set at *bt_level* that is assigned at *parent*[*i*].
exit, *exit*2, *exit*3	Flags used to control program flow.
flag[*i*]	An array of flags used to generate *new*, *old*, and *cross_ref*.
hold[*i*][*i*2]	An array that holds all possible values of *m*, neglecting the local rate term. For a given digit *t*, let $i = v[t]$. Then *hold*[*i*][0] is used if $x[t] = rword[t]$, and *hold*[*i*][1] is used if $x[t] \neq rword[t]$.
i, *i*2	Loop variables.
j	One of the parameters of the low-density code being used. The code is an (n, j, k) low-density code.
k	Another low-density code parameter. See *j*.
l	The current level in the codeword tree. The root node is at level zero, and the final level is *level_max*.
level_max	The final level in the codeword tree. *level_max* $= nj/k$.
m[*i*][*i*2]	Metric contribution for digit *i* when $x[i] = i2$.
met_bc	Stores the metric value of the best change discovered in Backtrack mode.
met_best	The greatest branch metric encountered in the current run of **Kernel**.
met_comp	When entering Backtrack mode, stores *met_total*[*bt_level*]. Future changes are compared to this quantity.
met_total[*l*]	Total path metric at level *l*. *met_total*[*l*] equals the sum of *metric*[*i*], from *i* = 1 to *l*.

met_val[*i*][*i*2]	The value of *met_total*[*bt_level*] that results after *branch*[*parent*[*i*]] is set to *i*2.
metric[*l*]	Branch metric at level *l*.
mode	Indicates the operating mode of the decoder. 0 = Forward mode, 1 = Backtrack mode, 2 = Reset mode.
n	The blocklength of the code.
n_parents	In Backtrack mode, the total number of parents of *bt_level*.
new[*l*][*i*]	The *i*-th digit in the n-set at level *l*.
old[*l*][*i*]	The *i*-th digit in the o-set at level *l*.
old_branch[*i*]	Stores *branch*[*parent*[*i*]].
old_parity[*i*]	Stores *parity*[*parent*[*i*]].
old_set[*i*]	Stores *set*[*parent*[*i*]][*parity*[*parent*[*i*]]].
p	The crossover probability of the channel.
p_flag[*i*]	A flag set to 1 if any branch setting is changed at *parent*[*i*].
par[*l*][*i*]	This array stores an ordered list of the parity checks that define the code. *par*[*l*][*i*] is the index of the *i*-th digit involved in the *l*-th parity check. Digits are numbered from 0 to $n-1$. The first parity check has index one, not zero, in order to agree with the numbering of the codeword tree levels.
parent[*i*]	In Backtrack mode, this array stores the locations of the parents of *bt_level*.
parity[*l*]	The parity of the o-set at level *l*. 0 = even, 1 = odd.
r[*i*]	The local code rate of the codeword tree. *r*[*i*] is the local rate at the level which contains digit *i* in its n-set.
rword[*i*]	This array stores the received word, obtained at the output of the communication channel. *rword* is a binary vector.
s, *s*2, *s*3	Integer temporary variables.
set[*i*][*i*2]	If $set[i][i2] \neq -1$, this indicates a decision by the decoder to assign $branch[i] = set[i][i2]$ when $parity[i] = i2$.
status	Records the output status of the decoder. 0 = Decoding completed, 1 = Decoding failure declared, 2 = Execution halted because *steps* > *step_max*.
steps	The number of forward and lateral moves performed by the decoder.
step_max	The maximum number of forward and lateral moves the decoder is allowed to perform.

stop	A flag set to 1 when the decoder is done.
sx	Floating point temporary variable.
$v[i]$	For *rword*, the number of violated parity checks involving digit i.
$x[i]$	The i-th digit of the codeword hypothesized by the decoder.

A.3.3 Storage requirements

Note: n, j, k are low-density code parameters
$k2 = 2^{k-1}$
$lm = level\_max + 1 = nj/k + 1$

int $a[k2][k-1]$, *active*, *b*, *b_flag*$[k][k2]$, *best*, *best*2, *branch*$[lm]$, *branch_best*, *branch_max*$[lm]$, *bt_level*, *cross_ref*$[n]$, $d[lm]$, *digit*$[k]$, *exit*, *exit*2, *exit*3, *flag*$[n]$, *i*, *i*2, *j*, *k*, *level_max*, *mode*, *n*, *n_parents*, *new*$[lm][k]$, *old*$[lm][k]$, *old_branch*$[k]$, *old_parity*$[k]$, *old_set*$[k]$, *p_flag*$[k]$, *par*$[lm][k]$, *parent*$[k]$, *parity*$[lm]$, *rword*$[n]$, *s*, *s*2, *s*3, *set*$[lm][2]$, *status*, *steps*, *step_max*, *stop*, $v[n]$, $x[n]$;

float *beta*, *hold*$[j+1][2]$, $m[n][2]$, *met_bc*, *met_best*, *met_comp*, *met_total*$[lm]$, *met_val*$[k][k2]$, *metric*$[lm]$, *p*, $r[n]$, *sx*;

Note: *m*, *met_bc*, *met_best*, *met_comp*, *met_total*, *met_val*, and *metric* must be able to store a value of $-\infty$. This can be implemented using a corresponding set of one bit flags.

A.3.4 The algorithm

Input	*n*; *j*; *k*; *par*; *p*; *rword*; *step_max*
Output	*x*; *status*; *steps*

Main program

```
Initialization;
stop = 0;
while (stop == 0) {

        if (mode == 0) {

                /* Forward mode. */
        while (l <= level_max and mode == 0) {
                **Kernel**;
                if (metric[l] >= 0)
                        l = l+1;
                else
                        mode = 1;
        }

        if (mode == 0) {                /* l > level_max */
                status = 0;
                stop = 1;
        }

}

else if (mode == 1) {

        /* Backtrack mode. */

        **Initialize backtrack**;
        while (any untried parents left) {

                **Choose next parent**;
                set[parent[active]][old_parity[active]] = -1;
                p_flag[active] = 1;

                while (there are any untried settings of active parent) {

                        **Choose next setting**;
                        if (parent[active] < l)
                                l = parent[active];

                        exit = 0;
                        while (l <= bt_level and exit == 0) {
                                **Kernel**;
                                if (l == parent[active])
                                        **Digit test**;
                                l = l+1;
                        }

                        if (exit == 0) {
                                s = branch[parent[active]];
                                met_val[active][s] = met_total[bt_level];
```

```
                    b_flag[active][s] = 1;
                }

                l = parent[active];

/* The statement above has an unexpected but useful effect. If a new parent
is chosen, and the new parent level > the old parent level, then the decoder
will first go to the old parent level. This allows the last change at the old
parent level to be reset as the decoder moves forward to implement the first
change at the new parent level. */

            }

            /* Return setting of active parent to previous value. */

            set[parent[active]][old_parity[active]] = old_set[active];

        }

        Decision;
        mode = 2;
    }

    else if (mode == 2) {

        /* Reset mode. */

        if (active != -1) {

            if (met_bc > met_comp) {

/* This implements the best change discovered in Backtrack mode. */

                set[parent[best]][old_parity[best]] = best2;
                l = parent[best];

            }

            else {

/* In this case, no change is accepted, and the decoder must reset
the last change. */
```

```
                        l = parent[active];

                    }

                    while (l <= bt_level) {
                        Kernel;
                        l = l+1;
                    }

                }
                mode = 0;
            }

    }
```

Initialization

```
s = n*j;
level_max = s/k;

/* Calculating level_max in two steps insures that integer division
will not cause truncation error. */

steps = 0;
met_total[0] = 0;
mode = 0;
l = 1;
for (i = 1; i <= level_max; i = i+1) {
    set[i][0] = -1;
    set[i][1] = -1;
}

/* Calculate new, cross_ref, old, d, branch_max. */
for (i = 0; i < n; i = i+1) {
    flag[i] = 0;
}
for (i = 1; i <= level_max; i = i+1) {
    s = 0;
    s2 = 0;
    for (i2 = 0; i2 < k; i2 = i2+1) {
        s3 = par[i][i2];
        if (flag[s3] == 0) {
            s = s+1;
```

```
                $new[i][s] = s3;$
                $cross\_ref[s3] = i;$
                $flag[s3] = 1;$
            }
            else {
                $s2 = s2+1;$
                $old[i][s2] = s3;$
            }
        }
        $d[i] = s;$
        if ($s <= 1$)
            $branch\_max[i] = 0;$
        else
            $branch\_max[i] = 2^{s-1}-1;$
}

/* Calculate $v$. */
for ($i = 0; i < n; i = i+1$)
        $v[i] = 0;$
for ($i = 1; i <= level\_max; i = i+1$) {
        if ($rword$ does not satisfy parity check $i$) {
            for ($i2 = 0; i2 < k; i2 = i2+1$) {
                $s = par[i][i2];$
                $v[s] = v[s] + 1;$
            }
        }
}

/* Calculate metric. */
$sx = (1-2p)^{k-1};$
$beta = (1-sx)/(1+sx);$
for ($i = 0; i <= j; i = i+1$) {

        $sx = p * beta^{j-2i} /(1-p);$

        /* With respect to the notation of Section 3.1,
            $P(y_t = x_t \mid V_t = i) = 1/(1+sx),$
            $P(y_t \neq x_t \mid V_t = i) = sx/(1+sx).$   */

        $hold[i][0] = 1 - \log(1+sx)/\log(2.);$
        $hold[i][1] = 1 + \log(sx/(1+sx))/\log(2.);$
}
```

```
s = 0;
for (i = 1; i <= level_max; i = i+1) {

    if (d[i] == 0) {
        s = s+1;
    }

    else {

        sx = d[i] - s - 1;
        sx = sx/d[i];

    /* Calculating sx in two steps insures the division
    is performed floating point, not integer. */

        s = 0;
        for (i2 = 0; i2 < d[i]; i2 = i2+1) {
            s2 = new[i][i2];
            r[s2] = sx;
            if (rword[s2] == 0) {
                m[s2][0] = hold[v[s2]][0] - r[s2];
                m[s2][1] = hold[v[s2]][1] - r[s2];
            }
            else {
                m[s2][0] = hold[v[s2]][1] - r[s2];
                m[s2][1] = hold[v[s2]][0] - r[s2];
            }
        }
    }
}
return;
```

Kernel

```
/* Kernel determines branch[l], and updates x, metric[l],
and met_total[l] accordingly. */

exit2 = 0;
exit3 = 0;

/* Control returns to the main program when exit2 is set to 1.
Setting exit3 = 1 causes the while loop to be executed exactly
one more time. */
```

```
/* Calculate parity[l]. */
s = 0;
for (i = 0; i < k-d[l]; i = i+1) {
        if (old[l][i] == 1)
                s = 1-s;
}
parity[l] = s;

/* Determine initial branch number. */
if (set[l][parity[l]] == -1)
        b = 0;
else {
        b = set[l][parity[l]];
        exit3 = 1;
}

while (exit2 == 0) {
        steps = steps+1;
        if (steps > step_max) {
                status = 2;
                stop = 1;
                return;
        }
        if (d[l] == 0) {
                if (parity[l] == 0)
                        metric[l] = 0;
                else
                        metric[l] = $-\infty$;
                exit2 = 1;
        }
        else {
                /* Generate branch. */
                for (i = 0; i < d[l]-1; i = i+1) {
                        s = new[l][i];
                        x[s] = a[b][i];
                }
                Assign x[s] for s = new[l][d[l]] so that
                parity check l is satisfied;

                /* Determine metric value. */
                metric[l] = 0;
                for (i = 0; i < d[l]; i=i+1) {
                    s = new[l][i];
                    metric[l] = metric[l] + m[s][x[s]];
                }

                /* Record best branch so far. */
                if (b == 0 or metric[l] > met_best) {
                    met_best = metric[l];
                    branch_best = b;
                }

                /* Decide whether to exit, and determine next branch number. */
                if (metric[l] >= 0 or exit3 == 1)
                    exit2 = 1;
                else if (b == branch_max[l]) {
                    exit3 = 1;
                    b = branch_best;
                }
                else
                    b = b+1;
        }

}               /* End while loop. */

branch[l] = b;
met_total[l] = met_total[l-1] + metric[l];
return;
```

Initialize backtrack

```
bt_level = l;
n_parents = k - d[bt_level];
met_comp = met_total[l];
active = -1;
for (i = 0; i < n_parents; i = i+1) {
    s = old[bt_level][i];
    parent[i] = cross_ref[s];
    old_parity[i] = parity[parent[i]];
    old_branch[i] = branch[parent[i]];
    old_set[i] = set[parent[i]][old_parity[i]];
    digit[i] = x[s];
    p_flag[i] = 0;
    for (i2 = 0; i2 <= branch_max[parent[i]]; i2 = i2+1)
        b_flag[i][i2] = 0;
}
return;
```

Choose next parent

```
$exit = 0$;
while ($active < n\_parents$ and $exit == 0$) {

        $active = active + 1$;
        if ($d[parent[active]] > 1$)
                $exit = 1$;

        /* If $d[parent[active]] <= 1$, then $branch\_max[parent[active]] = 0$,
        and so there are no new settings to try.  Therefore, a parent is
        chosen only if $d[parent[active]] > 1$. */

}
return;
```

Choose next setting

```
/* Chooses the next setting of $parent[active]$. */

$s = set[parent[active]][old\_parity[active]]$;

/* The following statement causes the decoder to skip the original setting. */

if ($s == old\_branch[active]$)
        $s = s+1$;

$set[parent[active]][old\_parity[active]] = s+1$;

/* The condition in the second while loop in the main program insures that
$s+1 < branch\_max[parent[active]]$. */

return;
```

Digit test

/* If the value of the digit with index given by *old*[*bt_level*][*active*] is not changed, then the current change at *parent*[*active*] will probably not raise the value of *met_total*[*bt_level*]. For this reason, the current change is aborted and the next change is tried. */

```
if (x[old[bt_level][active]] == digit[active])
        exit = 1;
return;
```

Decision

/* Determines the best change and decides whether to declare a decoding failure. */

```
met_bc = −∞;
for (i = 0; i < n_parents; i = i+1) {
        if (p_flag[i] == 1) {
                for (i2 = 0; i2 < branch_max[parent[i]]; i2 = i2+1) {
                        if (b_flag[i][i2] == 1 and met_val[i][i2] > met_bc) {
                                best = i;
                                best2 = i2;
                                met_bc = met_val[i][i2];
                        }
                }
        }
}
if (met_comp == −∞ and met_bc == −∞) {
        /* Decoding failure. */
        status = 1;
        stop = 1;
}
return;
```

REFERENCES

[1] E. Arikan, "An upper bound on the cutoff rate of sequential decoding," *IEEE Trans. Inform. Theory*, vol. IT-34, pp. 55-63, Jan. 1988.

[2] R. E. Blahut, *Theory and Practice of Error Control Codes*. Reading, MA: Addison-Wesley, 1983.

[3] G. C. Clark and J. B. Cain, *Error-Correction Coding for Digital Communications*. New York: Plenum Press, 1981.

[4] R. M. Fano, "A heuristic discussion of probabilistic decoding," *IEEE Trans. Inform. Theory*, vol. IT-9, pp. 64-74, Apr. 1963.

[5] R. G. Gallager, *Information Theory and Reliable Communication*. New York: John Wiley and Sons, 1968.

[6] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21-28, Jan. 1962.

[7] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: M.I.T. Press, 1963.

[8] I. M. Jacobs and E. R. Berlekamp, "A lower bound to the distribution of computation for sequential decoding," *IEEE Trans. Inform. Theory*, vol. IT-13, pp. 167-174, Apr. 1967.

[9] V. V. Zyablov and M. S. Pinsker, "Decoding complexity of low-density codes for transmission in a channel with erasures," *Problems of Information Transmission* (translation of *Problemy Peredachi Informatsii*), vol. 10, pp. 10-21, Jan.-Mar. 1974.

[10] V. V. Zyablov and M. S. Pinsker, "Estimation of the error-correction complexity for Gallager low-density codes," *Problems of Information Transmission* (translation of *Problemy Peredachi Informatsii*), vol. 11, pp. 18-28, Jan.-Mar. 1975.